Designing for Change: Reflective Metalevel Architectures for Deep Customisation in CSCW

Paul Dourish

Technical Report EPC-1993-102

Rank Xerox Research Centre
Cambridge Laboratory
61 Regent Street
Cambridge CB2 1AB

Tel:+44 1223 341500
Fax:+44 1223 341510

Designing for Change:
Reflective Metalevel Architectures for Deep Customisation in CSCW

Paul Dourish
Rank Xerox EuroPARC
Cambridge, UK[1]
dourish@europarc.xerox.com

Abstract

The past few years have seen a steadily increasing understanding of the need for customisation and tailorability in a range of computational systems. This has resulted both from a greater understanding of the role of individual and organisational work practices in human-computer interaction, and from the application of systems to more complex domains of activity. The increasing requirement that systems provide flexible and customisable behaviour, and that this customisation reach deeper into the system, forces a reconsideration of the techniques by which customisable systems are constructed.

This paper discusses an architectural approach which provides a means for building systems supporting deep customisation. The approach is based on providing an explicit reflective metalevel, allowing the design of computational systems with flexible and dynamic behaviours. This design approach has been used primarily in the area of programming language design; here it is considered in application to interactive systems more generally. In particular, an investigation of the use of this approach in designing flexible CSCW systems is described, and the wider implications of this approach for the notion of deep customisation are examined.

1. Introduction

Over the past few years, we have deepened our understanding of the nature of the interaction between individual and computer systems. Along with this has come a greater appreciation of the ways in which these interactions fit into a wider pattern of group interactions and work practices. One of the major consequences of this has been an increasing concern with issues of customisation. Traditional systems, designed around a fixed model of form and function, have been too enclosed and rigid to support the evolutionary practice of system use; and hence, we have become concerned with the ways in which systems can accommodate customisation and change. System customisation can be important for a variety of reasons, including the need to tailor systems to existing practice, to support temporal adaptation and evolution, to accommodate differences in individual styles, organisational roles and work processes, and to foster the ability to “own the technology”.

1. Also Department of Computer Science, University College, London.

As a result, many more information systems now provide facilities for individuals and groups to tailor and customise their interactions with the technology which supports them in a wide range of activities. However, support for tailoring activities is often quite limited. Frequently, users are presented with a set of controlling parameters which affect particular aspects of the system, and a set of allowed values. At best, this approach can be limiting; at worst, it is both limiting and, at the same time, overwhelming. This is the most that many everyday systems can provide.

A further step can be taken towards more tailorable design by allowing the users of the system to add new functionality and behaviours. Clearly, this is a considerably more complex activity, requiring more skill on the part of the user who perform the customisation; in exchange for this higher investment, a much more flexible system can be constructed. However, this approach can still result in a quite limited kind of customisation. Typically, the language in which customisation behaviours are expressed is rooted in the components of the application domain, and so the internal mechanisms of the system, by which the process of work is controlled, remain inaccessible and fixed.

This paper concerns the design, implementation and use of systems which are based around a notion of "deep" customisation, which can provide new kinds of control over systems in use. It focuses, in particular, on two major concerns in the design of customisable systems.

The first concern is primarily with the systems-level implications of customisation. The shift from rigidly-defined patterns of user/system interaction to a much more open, customisable approach forces a reconsideration of the mechanisms used in system implementation. While a certain amount of value can be (and has been) derived from the introduction of surface tailorability into otherwise fixed systems, a deeper notion of customisation requires a radical change in the way in which we design and structure systems. One goal of the work reported here is to understand how we can develop system architectures which support this deep customisation, and which can achieve more openness and flexibility in operation. In such systems, support for customisable behaviour becomes a central tenet of the design.

The second concern is the nature of deep customisation itself. The design approach advocated here is one which supports a form of customisation in which users can "reach in" to the system and make radical changes in the way in which tasks are supported. This deep customisation is essentially support for changes in *process*, rather than in *presentation*. This change is a dynamic process, in support of the mutual evolution of design and use. Our concern then, is how such deep customisation can be realised effectively, and what kinds of practices such flexible systems can support.

The principal focus of this paper is the use of *reflective metalevel architectures* as a mechanism for supporting this kind of flexibility in systems design. The essence of such architectures is the embodiment within a computational system of a self-representation which is amenable to examination (*introspection*, allowing the system to examine its own state at a particular moment), and change (*reflection*, allowing the system to modify its own behaviour in light of particular circumstances or decision). This approach derives originally from the work of Brian Smith [Smith, 1982; Smith, 1984; des Rivières and Smith, 1984] in the area of programming language semantics, but recently have been applied to other areas, including window systems design [Rao, 1991] and parallel programming systems [Rodriguez, 1991]. In this paper, we present this approach as a mechanism by which we can achieve deep customisation in information systems and, in particular, in systems supporting cooperative work.

The first section of this paper introduces the reflective metalevel approach, through an extended example. (The reflective metalevel design approach is technically complex; the purpose of this example is to explain and motivate the approach while maintaining an appropriately general perspective.) This example is the design of the Common Lisp Object System (CLOS), an object-oriented programming language. The constraints of this design effort requirement the developers to use a radical approach to the problems of customisation and flexibility, resulting in the specification of the CLOS Metaobject Protocol [Kiczales *et al*, 1991]. The CLOS Metaobject Protocol is an example of the reflective metalevel approach to design. I will show some of the problems which the developers faced, how the metaobject protocol provided an effective solution, and provide examples of the way in which the metaobject protocol can be used by CLOS users to customise the language.

The next section of the paper deals with flexibility in collaborative systems, and shows how we might apply the reflective approach, in much the same way as the CLOS developers did, to attack some of the critical issues in the design of CSCW systems. In particular, the focus of this discussion will be on the design of programming and design toolkits, where issues of generality are even more relevant. After this, I will go on to discuss issues in metalevel architecture design and deep customisation more generally, and point to some of the general research issues and open questions raised by this approach.

2. Customisable Language Design

Since the late 1950's, much work has been carried out in the design of computer programming languages. A programming language embodies a set of abstractions over the underlying hardware, and allows the programmer to express algorithms in terms of these (presumably more useful) abstractions. However, at first sight, it appears curious that, over 30 years later, new programming languages are still being developed. In fact, new programming languages appear with remarkable frequency—in the course of writing this paper, I have seen announcements of 3 different ones! Why is this? Surely, with 30 years' experience, we understand enough of how computer programs are specified and written to be able to produce the ultimate programming language for the range of tasks we perform?

Unfortunately, this just isn't the case, and there are various reasons for this. Programming language design exists in a tension between simplicity of expression, power of expression, conceptual elegance and efficiency of implementation. The hardware which programming languages "hide" simply works in very different ways from the ways in which we want to specify problems, and new types of problems demand new types of abstraction and expression. Furthermore, there is a problem with the notion of completely general abstractions. The details which the language's abstractions hide can often "show through" an implementation, and indeed are inherent in the process of implementation. So, even though a language might present a set of abstractions, the implementation of those abstractions might not efficiently support all expressions of programming ideas which can be constructed with them.

The programming language design community, then, is faced with a serious problem in pursuit of a flexible, truly general-purpose programming language. (Indeed, one school of thought within that community holds that a truly general-purpose language is not possible and should not be a design goal.) Recent work, focussing on the use of metaobject protocols, addresses this problem by providing a framework for flexible and extensible management of programming language

abstractions. The fullest expression of this principle is to be seen in the Common Lisp Object System (CLOS), a language for object-oriented programming in a dialect of Lisp.

2.1 Design Issues in CLOS

Lisp is one of the oldest programming languages in current use, its history stretching back to the late 1950's. Many dialects of Lisp have been proposed, developed and used over this period. In 1984, a definition was published for Common Lisp, a unification of the major dialects of Lisp, which quickly become a de facto standard for Lisp programming [Steele, 1984].

In 1986, a working group was formed to rationalise Common Lisp and produce a version suitable for standardisation by the American National Standards Institute (ANSI). One of the major extensions to Common Lisp which the ANSI group took on was the development of a language mechanism to support object-oriented programming.

Object-oriented programming is a style of programming which has long been used within the Lisp community; and, as with the Lisp language itself, a large number of implementations and dialects have been developed to support this programming style within Lisp. Indeed, a number of incompatible object-oriented dialects had been built on top of Common Lisp since the 1984 publication. The ANSI group therefore found themselves with a number of conflicting design goals.

1. As programming language designers, they wished to develop an elegant language, and one robust enough to stand as a national standard. This meant that the language had to be conceptually elegant, efficient in implementation on a wide range of systems, and support a wide variety of potential future uses.
2. As programming practitioners (and, for some, language vendors), they wanted to support the range of current programming practices employed either by themselves or by customers. This meant protecting the investment that customers had already made in particular object-oriented dialects based on Common Lisp.
3. As language standardisers, they wished to define a language which would be useful to a wide range of people, both current and potential future users of Lisp, in a single dialect. This included supporting the needs of the artificial intelligence, symbolic programming, language design and object-oriented database communities, amongst others.

The eventual solution to meeting these disparate goals was two-fold. First, the new language, known as the Common Lisp Object System or CLOS, was based on a unification of the two major dialects for object-oriented programming on Common Lisp (New Flavors [Moon and Keene, 1986] and Common Loops [Bobrow *et al*, 1986]). Second, the language was based on a mechanisms which allowed users of the system to "reach in" and change aspects of the programming language, specialising it for their own applications; this mechanism is called the *metaobject protocol*.

2.2 The CLOS Metaobject Protocol

The metaobject protocol (MOP) is a technique for designing and building extremely flexible, extensible systems, based on the combination of computational reflection [Smith, 1982] and object-oriented programming.

Reflection is the principle that a computer system embodies within itself a model of its own behaviour. This model can be examined by the system (*i.e.* it can examine its own state, or *introspect*),

and be changed to produce changes in its own behaviour (*i.e.* changes in the model are *reflected* back onto the system's behaviour). A metaobject protocol is a description of this model in object-oriented terms. This means that standard object-oriented programming techniques of abstraction, specialisation and polymorphism can be applied to the model, making it much more amenable to change.

In CLOS, the metaobject protocol is effectively a description of the CLOS implementation, written *as a CLOS program*, and accessible to the programmer developing applications in CLOS. The metaobject protocol provides a set of default behaviours which capture the base semantics of the object system, but allows for the programmer to change implementation or semantics for a particular application or situation. Because of the object-oriented nature of the metaobject protocol's specification, this does not mean the reimplementation of the whole language; rather, particular parts can be specialised and reused, so that the changes required to support new functionality are merely to those components of the system which are to behave differently.

2.3 A CLOS MOP Example

An example will serve to clarify both the nature of the problems which the language designers faced, and the way that the metaobject protocol can be used to solve them. First, though, it's necessary to set out some terminology and show the form of the CLOS metaobject protocol.

2.3.1 CLOS and OOP Terminology

Classes and *instances* are fundamental components of traditional object-oriented programming[2]. A class is a general description of a group of objects which a system will manipulate, and an instance is a particular item of data. Each instance is the a member of a class, and the class structure determines the internal components of the instance. In particular, it determines the internal named data values which an instance might hold. These are referred to as the *slots* of the instance. A programmer using an object-oriented language will use that language's mechanisms for class, instance and slot to structure the data for an application.

Along with each class description is a set of *methods* which describe the computations which the system should perform when a message is sent to an object. CLOS unifies the traditional object-oriented notion of a message with the Lisp style of functional programming by using *generic functions*; calling a generic function with an object as an argument is equivalent to sending a message to that object. Since generic functions, and the methods which implement them, are associated with classes, the same generic function call may result in different behaviours when called on different objects, since different methods may be called. When a generic function is called, the appropriate method or methods must be found ("*method lookup and combination*") and invoked ("*method dispatch*").

A given class is a *subclass* (effectively a specialisation) of some other class or set of classes. This subclassing provides the basis of an *inheritance* mechanism, which provides that all methods applicable a given class are also applicable to the subclass, unless overridden explicitly. So, all methods for the sample class **`vehicle`** are valid not only to objects of that class, but also to objects of its subclasses **`bicycle`** and **`automobile`**. However, the programmer may choose to

2. While the concepts are seen in almost all traditional object-oriented programming languages, the terminology varies. I use the CLOS terminology here.

provide new methods for the subclasses—for instance, the generic function which returns the number of wheels on the vehicle should probably be redefined for the subclasses.

2.3.2 The Form of the CLOS MOP

The form of the CLOS MOP is a description of the components and behaviour of the system in terms of these object-oriented mechanisms; that is, the MOP specifies the behaviour of CLOS as a CLOS program. The fundamental components which CLOS manipulates (such as classes, generic functions and methods) are described as CLOS objects themselves; and the operations which make up the behaviour of the object system are defined as generic functions, with defined methods providing the default behaviour. In addition to the standard programming interface, with provides mechanisms for using the object system (*e.g* defining classes and creating instances), CLOS also offers these metaobject protocol components to the programmer.

The programmer, then, can use the standard object-oriented programming techniques to specialise and extend the behaviour of the object system itself. A new kind of class with a different sort of behaviour can be defined by creating a subclass of the standard "class" representation, and specialising the methods which are to change. The object-oriented nature of this structure means that we can change the system's behaviour *incrementally*; the programmer need only change those functions which are related to the desired new behaviour, and all others will be inherited from the original class description automatically.

2.3.3 Example: Instance Representation[3]

Consider two sorts of systems which might be constructed with CLOS. The first is a graphics system, which require a class called "point" to describe points on the graphics display. The structure of class "point" is quite simple—it specifies that its instances should have just two slots, called "x" and "y". The second is a knowledge representation system, which will use the class structure to represent data about people. Class "person" is much more complex. At various points in the program, we may need slots for hair colour, height, age, weight, and many other characteristics.

These two applications present a problem to the language designer or implementor. While both can be represented in terms of class and object abstractions, the detail of how these abstractions should be managed to meet the requirements of the applications are very different. The graphics system may have very many of the point objects, and require extremely fast access to the slots (*e.g.* to support real-time user interactions with the graphic display). This suggests that the object representation should be small, with the slots located at fixed, known points in the structure. However, this approach will be extremely inefficient for the second case. Fixed allocation for each slot of the person objects will result in huge, unwieldy objects containing sparse data, with much wasted space. For this case, a *lazy* allocation mechanism, which only creates slot storage as needed, would be much more appropriate. One fixed implementation of object representation will not suffice for efficient support of both these examples.

This is where a metaobject protocol for the object system can be employed. The object system provides a default implementation of classes and object representation, in the predefined class **`standard-class`**. In a given implementation, any particular technique might be used—let's assume that it's the fixed slot allocation technique which can be used for the class "point", so that our first

3. This example is due to Gregor Kiczales.

example system can be supported by the object system. The metaobject protocol allows the programmer to modify parts of the object system to support efficient implementation of the second example, class “person”. The programmer would create a new metaclass[4] called **`lazy-class`** which will support the lazy slot allocation technique; **`lazy-class`** would be a subclass of **`standard-class`**.

For this new class, then, the programmer can define a set of MOP methods which override the default slot allocation mechanism. In particular, changes need to be made to the instance allocation strategy, and to slot read and write. The metaobject protocol defines the generic functions which are called in order to perform these tasks, and so new methods for these generic functions, defined on class **`lazy-class`**, will override the default mechanisms. Since **`lazy-class`** is a subclass of **`standard-class`**, the default behaviours will be inherited for all class-related behaviour which is not specifically overridden.

This simple example serves to illustrate the way in which the metaobject protocol is constructed and used. Effectively, it provides the CLOS programmer with a window onto the implementation; an open window, which allows the programmer to reach into the implementation and make changes, within certain constraints. The metaobject protocol allows CLOS, and any given implementation, to be applied to a much wider range of applications than would be possible otherwise; for where the implementation decisions made by either the language designers or implementors would make it difficult or impossible to apply the language’s abstractions to some situation, the metaobject protocol provides the means by which the programmer can revise those decisions. It would be hard or impossible for a compiler to automatically perform this sort of optimisation, since it is based on the knowledge which the programmer can supply on how the classes being optimised will be used. A compiler could not guarantee, from a static analysis of the program, that these optimisations are appropriate—and yet, without them, some class of applications cannot be reasonably constructed. Clearly, the metaobject protocol is a powerful tool; great care must be taken in its design and use, and we will return to these issues later. However, it also provides a great deal of flexibility in systems design, and provides a novel view of customisation.

3. Wider Issues in Customisation

The CLOS metaobject protocol was one of the earliest large-scale efforts of metaobject protocol design, and remains the largest and most wide-spread such effort to date[5]. It reflects the basis of this work in programming language design and implementation; indeed most other metaobject protocol development efforts have remained rooted in related programming language issues, including other flexible languages specifications such as the EuLisp metaobject protocol [Bretthauer *et al*, 1992; Padget and Nuyens, 1992], parallel programming systems [Rodriguez, 1991] or flexible compilation strategies [Lamping *et al*, 1992; Ashley, 1992]. However, other work has demonstrated the value of the reflective metalevel approach to other areas of system design. The most notable example is Silica [Rao, 1991], a reflective architecture for window systems. Silica is based on a notion of *implementational flexibility*, which provides access to a reflective metalevel without the metacircular[6] approach common in other systems.

4. In many object-oriented systems, classes are realised as objects. A metaclass is the class to which a class object belongs; it is the class of a class. Standard-class is a metaclass; it is the class to which user-created classes belong by default.

5. Common Lisp and CLOS are approaching official ANSI standard status. While the form of the metaobject protocol will not be specified, all major Lisp vendors have agreed to work with the metaobject protocol specified in [Kiczales *et al*, 1991].

Essentially, Silica provides an “existence proof”, showing that the reflective metalevel approach can be applied to systems other than programming languages and can provide a means for flexible, open systems design in other application domains. It demonstrates the value of the notion of opening systems for introspection and reflection in avoiding implementation trade-offs in system design and providing, in a toolkit, a basis for building a much wider range of systems than is traditionally the case. The reflective metalevel provides the mechanisms through which designs can be tailored, customised and extended in order to support novel styles of interaction and implementation-level support for a variety of differing applications or user communities.

This opens the question of how we might apply the reflective metalevel approach to the design of toolkits and systems in other areas of system design where flexibility issues are important; human-computer interaction and information systems design are clearly domains where this approach could be extremely valuable. The power of this approach is in the ability to move design decisions “down-stream”; the decisions which implementors make can be examined and revised at later points in the process, allowing application developers or end-users to adapt systems to their own circumstances and requirements. Reflective systems, essentially, are designed around the principle that they will be subject to such customisation; from the ground up, they are designed for change.

The research described in this paper, then, is an investigation of the way in which reflective metalevel architectures can be used in a variety of interactive applications, and can provide a structure which accommodates the requirements for flexibility and openness which we find in these environments. In particular, it is concerned with the design of systems which support cooperative work, which highlight these issues. In the next section, then, we will investigate some of the flexibility issues which arise in collaborative systems, and examine how a CSCW architecture based on metalevel processing could provide a means for building more open and malleable systems.

4. CSCW and Reflective Design

The domain of computer-supported cooperative work (CSCW) is one where flexibility demands are particularly critical. It moves beyond the traditional domain of supporting the interactions between a single user and a computational system to supporting the interactions between different individuals and groups. This introduces even more degrees of freedom, and brings us more directly into contact with the organisational factors which affect human-computer interaction.

In particular, we can highlight three areas of flexibility required of CSCW systems:

1. *Static* flexibility is a form of flexibility in a system which does not change during a work session, or changes only slowly from session to session. The usual single-user issues of customisable interfaces are still present in CSCW applications, and fall into this area, but their multi-user nature makes these issues more complex. First, individual customisation activity, for instance, must be coherent across the group, and the system must be able to support this. Second, systems must support customising activity by the group as a whole, to accommodate different working styles adopted by individuals, groups or organisations. (Some of these issues are described in [Greenberg, 1991].) Third, the importance of being able to adapt CSCW systems

6. Lisp interpreters are typically implemented in the same dialect of Lisp that they interpret. Such an arrangement is called a *metacircular interpreter.* CLOS, by being defined in terms of a CLOS program, also displays this metacircularity.

for compatibility with existing single-user tools employed in the same environment is perhaps even more critical for CSCW systems, since group members may well use different single-user tools.

2. *Dynamic* flexibility involves responding to changing aspects of the group and the group's behaviour. Observations of group behaviour reveal the extent to which it will change in the course of a collaboration, and how such changes are frequently swift and tacit. The activity of individual participants also changes as group work progresses, as roles and responsibilities are renegotiated, and as members switch between individual and group work. Group membership will also change, with individuals joining and leaving the group. Acknowledgment of this dynamic element is often lacking in existing CSCW systems, which provide support for one particular style of work, or one particular activity within a collaboration. Inflexible support for group dynamics, and its impact on the collaborative process, are problems cited in studies of computer-supported collaboration [*e.g.* Austin *et al*, 1990].
3. *Implementational* flexibility covers the adaptability of a system to a variety of underlying infrastructural and architectural changes. For instance, the network architecture might dictate a choice of centralised, distributed or replicated architecture for the shared application; and this might need to change with the dynamics of the group. So, not only would we like a system to be able to support all of these approaches, depending on particular circumstances, but we would also want the system to be able to change its policy for data distribution as the collaboration progresses. While these changes may often be minimal on the surface (*i.e.* at the interface level), they are major reorganisations of the internal state of the system.

These are areas of flexibility which we would hope to see addressed in an application if we wished it to be widely applicable to a range of collaborative working situations and environments. However, much as was the case when we considered language design, some of the flexibility requirements faced by the designers of toolkits are even more difficult to resolve than those faced by applications designers.

4.1 Toolkit Flexibility

The designer of a toolkit is faced with higher-level flexibility problems. Since the toolkit is designed to be used in more than one application, the flexibility demands of each different application area must be reflected, in some way, in the design of the toolkit. Indeed, just as an application designer may well not be able to foresee the way in which an application will be used, the toolkit designer may not even know what applications will be generated from the toolkit. Flexibility, then, is the essence of the toolkit designer's task.

The goal of the toolkit is to provide basic facilities which can be combined in a number of ways to produce different applications. Typically, toolkits for user interface or CSCW design will offer a variety of fundamental toolkit components (or "widgets"), and probably some different ways of combining them to form higher-level units and applications. Flexibility in the toolkit requires that the toolkit components be flexible, as well as the mechanisms by which they are combined. Yet these components must all work together when combined in an application. To an extent, this is achieved by any existing toolkit design; however, as in programming language design, the traditional mechanisms of software construction make deep changes very difficult.

By analogy with the three areas of flexibility described above, we can consider three examples of flexible system behaviour which have serious implications for the kinds of flexibility which is required at the toolkit level.

4.1.1 Example: Object Locking

Many CSCW systems operate, at least metaphorically, through shared access to a workspace containing objects which can be created, arranged, edited or deleted by each user. Many such shared workspace systems embody a notion of "object locking" which prevents one user from making changes to an object while another user's changes are in progress. We could consider, in fact, that all shared workspace systems use locks on objects, even in the degenerate case where those locks are "null" (*i.e.* a lock request will never be denied). Object locks, then, seem like the type of abstraction which a toolkit for CSCW applications might well provide.

This leaves the toolkit designer with the decision as to how the toolkit-provided locks should work. If the toolkit locks are degenerate locks, then they will not directly support applications where "strong" locking of data is required. Such applications might include collaborative CAD systems in which the output will be used to generate instructions for a milling machine—errors caused by inconsistent data which was not adequately locked could prove costly and dangerous. However, if the toolkit locks are "strong" locks, in which, say, the process of moving a lock from one user to another might involve an explicit action to request the lock and an explicit action to relinquish it, the collaborative CAD system will be more directly supported, but other applications will be excluded. For instance, a simple shared electronic whiteboard system for use in brainstorming meetings would not be well supported by strong, explicit locks, which would interfere with the brainstorming process.

Clearly, then, the designer of the toolkit has a difficult decision. To support one kind of lock and not others would be to limit the applicability of the toolkit; it could only be used to build certain applications. The designer, then, might choose to supply both types of locks, either as separate components or through some mechanism for parameterised locking—but what is to be done in the cases which require a third type? There is a twin problem. First, each new lock type that the designer provides in the toolkit increases the complexity of the toolkit, since each lock much interoperate with each other one, and with each shared data type. Second, the designer can never provide enough locks; there will always be difficult applications which require a new locking strategy. We've already included null locks and explicit-grant/explicit-release locks, but what about multi-user locks, tickle locks, implicit-grant locks, and all the others? In a conventional toolkit, an application developer who wants to use a new locking strategy has no clean means to privately implement the new mechanism and make it work with the rest of the toolkit. The toolkit is "closed" to extensions of this sort.

A reflective toolkit, based on a metaobject protocol, takes a very different approach. It provides a protocol for locks—that is, it reveals the mechanisms within the toolkit by which locks are obtained, managed, accessed and released. What's more, it provides the toolkit designer with the means for allowing the user to provide new locking strategies. This extension is provided through the standard object-oriented techniques of subclassing and incremental specialisation. The metaobject protocol defines the basic objects of our system, the operations which the system can perform upon them and hence the relationships between them. In this case, locking is an operation which is performed upon *shared objects* and perhaps *users*, metaobjects of the system. Thus:

1. The metaobject protocol specifies the a generic function **`obtain-lock`** which specialises on users and objects. With this, the programmer, using the toolkit, can define different locking actions for different combinations of users and objects.
2. The programmer can create subclasses of the shared object class called **`strong-lock-object`** and **`null-lock-object`**. Then, methods can be defined for the obtain-lock function for each of these classes, with different behaviours. For instance, objects of class **`strong-lock-object`** will have an explicit grant/request behaviour; objects of class **`null-lock-object`** will have no locking (*i.e.* a lock request will succeed immediately). New methods can be defined which provide new locking strategies as are appropriate for the particular application under consideration.
3. Similarly, we can create subclasses of the *user* class with different lock behaviours. These might correspond to the roles the users play at any given point in the collaboration. For instance, we could define methods so that **`obtain-lock`** will always fail for users who are "readers" but not "writers", so that they cannot obtain the lock they need to change an object.

With this mechanism, then, we can also achieve new behaviours which we hadn't considered before. Different locking strategies can be used for different objects which exist within the same shared workspace, or for different users, or at different times in the course of a collaboration. The applications programmer has a mechanism for reaching into the toolkit and providing support for new strategies appropriate for particular situations, and so a much wider range of systems can be constructed.

4.1.2 Example: Degree of Sharing

In looking at collaborating groups, one common observation is the range of mechanisms which groups can use to coordinate their activity. These are not variations from group to group, but will frequently occur within a single group and a single collaboration. For instance, even within the course of a single meeting, a group may adopt a wide range of coordination policies to govern their interactions—formal agenda following, rote floor control, informal brainstorming, subgroup or individual working, and so forth. Frequently, the shifts between these working modes will be swift and tacit. Frequently, the nature of the groups' interaction through a CSCW tool will be fixed through the particular sharing mechanisms which the tool embodies. For instance, tools which share access to a group workspace at the level of whole documents, paragraphs, views or windows, carry with them implications for the ways in which they can be used to support group activities, and, in turn, the kinds of mechanisms which can be used to regulate those activities. In such a fixed system, the result is that a group must either adapt its style to suit the tool, or use the tool merely for particular parts of the collaboration, and hence deal with the difficulties of supporting other parts, or switching between different tools.

What we would like to do, then, is to provide mechanisms by which different degrees of shared access can be defined within a given application, and control this dynamically so that users can move between the different support and coordination mechanisms. Typically, the degree of sharing which can be supported within a system is constrained by a number of architectural factors including the user interface and data distribution mechanisms. These components, being internal to the system, are not accessible to the toolkit user, and certainly are not available for manipulation at run-time. However, a reflective metalevel architecture provides a framework in which access to these components can be made available and hence a toolkit can support the flexible behaviour which is required in this case.

One mechanism which could be used to provide this flexibility is to generalise the notion of the "shared workspace". Consider the definition of a shared workspace to be those data components which are common to the view of all participants. From the usual perspective of CSCW design, those data components will be domain-level items of information shared by the group--text in the case of a shared document editor, graphical objects for shared drawing programs, and so on. However, in the reflective approach, items of "internal" information, referring to system configuration and implementation, can also be manipulated as data components in themselves. Thus, a reflective system could also make components of the user interface, for example, into shared objects by placing them in the shared workspace. In object-oriented programming terms, this might be achieved by making those components inherit the locking, distribution and consistency behaviours of shared objects. Using this approach, then, the mechanisms which allow objects to be moved into and out of the shared workspace can also be used to allow the toolkit programmer to control the degree of sharing which the tool should provide at a given time. This control is intrinsically dynamic, as it derives from the dynamic properties of the shared workspace itself; hence, the extent of sharing within the application can change in the course of the collaboration, or even under the control of collaborating users. Thus, the system allows customisation of the mechanisms used to support the collaboration, allowing it to be used in a wider range of circumstances, by different group with different coordination strategies, or by groups using different strategies at different stages of a single collaborative activity.

4.1.3 Example: Data Distribution

A third example will serve to show the way that a metaobject protocol can provide flexibility at the implementation level. In considering a system description at this level, it is important to bear in mind the impact of these implementation features on the upper layers of the system and, ultimately, in the kinds of practices and user interactions which can be offered by the system as a whole.

We have taken the perspective of looking upon CSCW systems as modelling notional shared workspaces. At the implementation level, then, we must consider how these workspaces are to be realised. This raises a number of issues concerned with data distribution. How are the data structures which represent information within the shared workspace distributed across the multiple computers which provide views into this space?

CSCW systems have traditionally chosen one of two approaches. The *centralised* approach is to maintain the representations of shared data at one site on the network, and, at this side, to provide a client/server style interface to the shared space, with facilities for data retrieval and editing. Thus, in order to view the space, a user's machine will make queries of the server holding the information, which will respond with lists of available objects and attributes. In order to make a change, a user's machine may request the server to lock some data, and then send an update message to make the change. While the details of the client/server interaction are "hidden" by an interface which may, for instance, use direct manipulation techniques to create the illusion of direct interaction with local data, circumstances can arise where this illusion breaks down. Indirect manipulation of shared data requires fast access to the server which holds it. In the case of interactions over long distances, or sharing of data between a very large number of individuals, the centralised approach may interfere with easy and natural interaction with the data through the introduction of network or server delays.

The other common approach is the *replicated* approach, in which each participating user in a collaboration has a private copy of the shared information. In this case, queries and updates can be

performed locally, without requiring network interaction, and hence without incurring the costs of potential network delays. However, in discharging the problem of fast access, the implementor has created the new problem of maintaining data consistency in the light of potentially conflicting changes occurring simultaneously to different copies of the data. The nature of the conflict resolution mechanisms which can be adopted will typically depend on the domain.

Although these are the two most common architectures for data distribution, others are clearly possible. A *migratory* solution would involve elements of shared data which move between one machine and another, perhaps always being local to a machine where they are to be changed. A partially-replicated approach involves multiple copies of data which may dynamically be created and merged in response to system usage, and which may also be migratory.

Clearly, then, the choice of data distribution strategy depends on a number of factors, including the expected usage patterns of the system, the tasks being supported and the network infrastructure available. Indeed, a single strategy may not be appropriate for all the data in the shared workspace, or for a single shared item at different points in the collaboration.

The metaobject protocol approach, then, makes explicit the strategy by which data items, or classes of data items, will be distributed; and provides a mechanism by which distribution decisions can be made, and distribution-related functionality attached. This can be extremely powerful in a number of ways. Statically, it allows the toolkit to be applied to situations in which the network facilities are quite different; and thus, a single system can scale much more easily since it has a range of distribution strategies available. Dynamically, it means that a system could adjust the data distribution policy adaptively, in response to the demands of the task being performed, the dynamics of the group membership, or the patterns of activity of the members of the collaboration.

4.2 The MOP Approach

Having seen examples of the way in which a reflective metalevel architecture, as realised by a metaobject protocol, can allow the construction of much more flexible systems, how can we characterise the MOP approach?

The metaobject protocol is an interface to a reflective, metalevel description of system behaviour. The description is at the metalevel because it describes the behaviour and organisation of the system itself, rather than the application domain. It is reflective in that it is amenable not only to introspection, but also to change which will be reflected in the subsequent behaviour of the system. The metaobject protocol is an object-oriented description, and the standard techniques of object-oriented programming can be applied to its use.

The use of object-oriented techniques in the metaobject protocol is a key element in making the reflective metalevel description usable. Firstly, through the use of class-based polymorphism, it means that multiple implementations and behaviours for particular components of the system can peacefully coexist. This means that the system designer can supply default behaviours, and the user can create new behaviours for particular situations, with those new behaviours being used only where appropriate. Secondly, through the use of specialisation, the system can support incremental change. Creating a new system behaviour does not imply beginning from scratch; rather, the new behaviours can be specified as incremental changes from existing system behaviours and defaults.

The design of the metaobject protocol, then, is essentially about designing a framework in which appropriate changes can be specified. The role of this framework is to guide subsequent changes, and also to make guarantees about the interaction of components, whether at the base-level or the meta-level, and whether system supplied or customised by users.

5. MOP Design Issues

The preceding sections have outlined the use of reflective metalevel architectures, and shown how a metaobject protocol, as an interface to a reflective representation of a system, can be used to provide a mechanism by which the users of a system (programmers, in the case of a toolkit or language) can "reach in" and make changes to that system, in order to achieve more appropriate or efficient support for a particular application or domain. One issue which has not been dealt with, however, is the problem of *designing* such a protocol.

Along with the power which reflective metalevel architectures offer comes a great potential for accidental misuse. The ability to reach in and alter the implementation strategies of a programming language, for instance, is less than entirely useful if my metalevel modifications cripple the performance of the language. To quote Foote, "*Existing programming languages already give clumsy programmers more than ample opportunities to shoot themselves in the foot... Reflective systems allow the clumsy programmer to fire several rounds per second into his foot, without reloading*" [Foote, 1992]. As a result, a major design goal of a metaobject protocol must be to structure the interface so as to *guide* the user in making appropriate modifications.

Based on the experience of the CLOS MOP and others, Kiczales [1992] has described four properties of metaobject protocols which are important in this regard:

1. *scope control* gives the programmer using a metaobject protocol control over the scope of changes that are made. The metalevel should be constructed so as to minimise potential unseen implications of local changes.
2. *conceptual separation* allows the programmer to change some aspect of an implementation through the use of one part of the metaobject protocol, without having to understand the whole metalevel interface. This criterion encourages, as far as practical, the decomposition of a protocol into sub-protocols dealing with particular aspects of the metalevel interface.
3. *incrementality* is the property which allows behaviour to be customised by changing merely those elements which are to behave differently. It should not be necessary to reimplement the whole system, but simply to modify particular components as necessary.
4. *robustness* refers to the system's ability to limit the impact of user errors, and to maintain some kinds of guarantees about performance in the face of open-ended user extension.

These are general design principles for a reflective metalevel architecture which make that architecture easier to use. The two issues are, firstly, to make it clear to the user of a metalevel interface what kinds of changes and extensions are appropriate, and secondly, to make it as simple as possible to make those changes; these design principles address both issues to some degree.

Above all, in attempting to guide appropriate use of the metaobject protocol, it is critical to provide the user with an understanding the *design space* which the protocol represents. The default behaviour is a point in that space, and the metalevel interface provides the means to move around in it. Thus, the default behaviour of the system expresses a position, presumably reasonably central, within that space, and so, from the default behaviour and the structure of the interface, the user will

derive certain characterisations of the space which will guide exploration through extension and change. The user's understanding structure of the space may also be informed by an understanding of the domain of application—in the case of CLOS, for instance, an understanding of the problems and trade-offs involved in the design of object-oriented programming systems. The rooting of a metaobject protocol design in a particular domain, and even in a particular set of problems and trade-offs, becomes an important component in the design of an effective and usable metaobject protocol.

In considering the use of a reflective metalevel for customisation in collaborative settings, it is also important to consider customisation as a collaborative process in itself. Mackay [1990a] has shown the way in which customisation of software is a complex collaborative process in which customisations are shared, exchanged and modified within a work group; Nardi and Miller [1990] have also shown that collaborative problem-solving in programmable software such as spreadsheets is the norm, rather than the exception. Clearly, in considering customisation behaviour in work groups, we must consider this aspect of customisation activity. Maclean *et al* [1990] describe their experiences with Buttons, customisable encapsulations of Lisp programs, and point to the importance of a "tailoring culture", encouraging experimentation, customisation and sharing. Clearly, there are important lessons from studies such as these, building on the principles outlined above, for the appropriate structuring of customisation facilities for cooperative customisation activity.

In general, many of the issues in metaobject protocol design remain open problems. Kiczales' design principles and the notion of exploration of a design space provide pointers, but few solutions. Design approaches evolve as our experience with metalevel reflective architectures increases; existing usable metaobject protocols show that, at least within certain domains, solutions exist. We shall move on, then, to examine the nature of customisation in reflective systems.

6. Deep Customisation

This paper has outlined the way that systems developers and users can use the power of reflective metalevel architectures to make information systems and toolkits more open, flexible and customisable than traditional systems. Reflection provides an architecture for customisation; the design of reflective systems is, from the very basis of the implementation, design for change. Customisation, extension and dynamic change are the essence of the reflective metalevel approach.

This design principle changes the nature of customisation in information systems. Not only does it produce systems in which customisation is supported as a primary activity, but it also provides a model of *deep* customisation. Deeply customisable systems are ones in which the user is empowered to make deep structural changes, rather than simply surface modifications in the interface. Fundamentally, it is a model of customising *process*, rather than *presentation*. Work such as Mackay's study of user innovation in the use of software systems [Mackay, 1990b] points to the way in which users will adapt the way in which they use software in order to support processes and work practices; reflective systems provide a way to change not only the way in which the software is used, but the nature of the software system itself.

These architectures, then, provide a means to extend the notion of customisation in software systems. Even the "radical tailorability" of Oval [Malone *et al*, 1992] is primarily based on a model of flexible information *presentation* and works on the assumption that the user can find the appropriate way to use the tool in order to support some set of practices or work goals. In reflective systems, the link between flexible information tools and flexible work practices can be made much

stronger, through the ability of the information system to adapt to the practices which it supports, even as those practices develop and evolve. Its helps to bridge the gap between information systems implementation and the participative and evolutionary models of software design; indeed, analogies have been drawn between aspects of participative design and the work on the CLOS MOP [Trigg, 1992].

In providing facilities for deep customisation, whether used by the applications designer or, in some form, the end user, it is critical that these facilities be manageable and usable. The critical element here is the relationship between reflection and object-oriented programming, as realised by the metaobject protocol. Here, the standard techniques of object-oriented programming can be used to make access to the reflective metalevel much simpler. The most important aspects of this relationship are, first, the use of *subclasses* and *specialisation* in providing mechanisms for incremental change, and, second, the use of *polymorphism* to allow multiple representations and behaviours to peacefully coexist in a reflective software system.

It is important to point out that customisation has been discussed here on two levels. The first is the use of a customisable toolkit by an applications developer, affording the opportunity to tailor and restructure the toolkit to suit particular application domains and revise implementation trade-offs which the toolkit developer has made in the course of realising the toolkit's abstractions. The second is customisation by end users to change the nature of their interactions which a system constructed with a reflective metalevel. This second form of reflective customisation, clearly much more intricate and problematic, is part of a longer-term research plan. In the first instance, we are concerned far more with the use of reflection in providing flexible toolkits. Building on this, we are working towards a third, intermediate use of reflection, in which a reflective toolkit can be used to build systems which are themselves reflective, and in which the applications designer can exploit the reflective nature of the application in order to provide support for the dynamics of group behaviour.

At each of these levels, though, the use of reflective metalevel architectures provides a new way to provide tailorability in information systems, and in particular, to provide a form of deep customisation concentrating on process rather than presentation. Based on existing experiences with customisable systems, and studies of innovation in the use of software systems and their interactions with work practice, the reflective metalevel approach has great potential in linking the practice of customisation to the design of interactive systems, and in blurring the hard distinction between customisation and development.

7. Conclusion

This paper has outlined an approach to the design of customisable computer systems. Reflective systems embody a representation of the system's behaviour within the system itself, causally connected to the behaviour which it describes, and amenable to examination and change. Thus, the architecture of the system is based around an appreciation of the need for customisation and extension. The customisation which it embodies reaches deep into the system, providing for tailorability not only of information presentation, but also of the process of work and manipulation surrounding the information and interactions which the system supports. This is a novel application of the reflective techniques, which have primarily been applied in the area of programming language design and implementation; it is also a new model for the provision of customisability in interactive systems design. Reflective systems design is, in essence, design for change.

This approach provides a link between, on the one hand, systems architecture, design and implementation, and, on the other, the lessons which have been learned in the areas of customisation and innovation in system use. Such a link is especially important in dealing with dynamic and evolving systems such as those supporting collaborative working and work group interaction. The reflective metalevel approach can be provide customisable behaviour at a variety of levels. Our primary research goal concerns the use of reflection in a flexible CSCW toolkit, although there are clearly important implications for deeply customisable end-user systems.

It is clear that reflective metalevel systems design as an approach to customisation raises a number of serious issues, especially in terms of the degree of sophistication required in order to make useful and productive customisations. These issues are being approached in stages, by first dealing with toolkit design and then moving towards a model of "end-user design" in applications. Even in toolkit design, however, the reflective model has great potential in allowing a toolkit a much wider range of applicability than traditional techniques; the experiences of metalevel interfaces and metaobject protocols in programming languages demonstrate the value of this approach in a similar area. If appropriate ways can be found to harness the power of the reflective approach to systems design, then systems which can adapt or be adapted to suit evolving work practices may be within our reach and shed new light on the nature of customisation in computer systems.

Acknowledgements

I am grateful to a number of colleagues, including Hal Abelson, Danny Bobrow, Austin Henderson, Gregor Kiczales, Jim des Rivières and Luis Rodriguez, for extensive and fruitful discussion of these ideas. Jon Crowcroft, Rachel Hewson and Wendy Mackay have also made helpful comments on earlier drafts of this paper. Of course, I am responsible for remaining mistakes, irrelevancies and general weirdness.

References

[Ashley, 1992] Michael Ashley, *"Open Compilers"*, Xerox PARC Technical Report, to appear.

[Austin *et al*, 1990] Laurel Austin, Jeffrey Liker and Poppy McLeod, *"Determinants and patterns of control over technology in a computerised meeting room"*, Proc. ACM Conference on Computer-Supported Cooperative Work CSCW '90, Los Angeles, CA, October 1990.

[Bretthauer *et al*, 1992] Harry Bretthauer, Harley Davis, Jürgen Kopp and Keith Playford, *"Balancing the EuLisp Metaobject Protocol"*, Proc. ISMA'92 International Workshop on Reflection and Metalevel Architecture, Tokyo, November 4–7, 1992.

[Bobrow *et al*, 1986] Daniel Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik and Frank Zbydel, *"CommonLoops: Merging Lisp and Object-Oriented Programming"*, Proc. ACM Conf. Object-Oriented Programming, Systems, Languages and Applications OOPSLA '86, Portland, OR., September 1986.

[Foote, 1992] Brian Foote, *"Objects, Reflection and Open Languages"*, Proc. ECOOP'92 Workshop on Object-Oriented Reflection and Metalevel Architectures, Utrecht, Netherands, June 1992.

[Greenberg, 1991] Saul Greenberg, "*Personalisable Groupware: Accommodating Individual Roles and Group Differences*", Proc. European Conf. on Computer-Supported Collaborative Work ECSCW91, Amsterdam, September 1991.

[Kizcales *et al*, 1991] Gregor Kiczales, Jim des Rivières and Daniel Bobrow, "*The Art of the Meta-Object Protocol*", MIT Press, Cambridge, Mass., 1991.

[Kiczales, 1992] Gregor Kiczales, *"Towards a New Model of Abstraction in Software Engineering"*, Proc. IMSA'92 Workshop on Reflection and Metalevel Architectures, Tokyo, Nov 4–7, 1992.

[Lamping *et al*, 1992] John Lamping, Gregor Kiczales, Luis Rodriguez and Erik Ruf, *"An Architecture for an Open Compiler"*, Proc. IMSA'92 Workshop on Reflection and Metalevel Architectures, Tokyo, Nov 4–7, 1992.

[Mackay, 1990a] Wendy Mackay, "*Patterns of Sharing Customisable Software*", Proc. ACM Conference on Computer-Supported Cooperative Work CSCW '90, Los Angeles, October, 1990.

[Mackay, 1990b] Wendy Mackay, *"Users and Customisable Software: A Co-Adaptive Phenomenon"*, PhD Thesis, Sloan School of Management, MIT, Cambridge, Mass., 1990.

[MacLean *et al*, 1990] Allan MacLean, Kathleen Carter, Thomas Moran and Lennart Lövstrand, *"User-Tailorable Systems: Pressing the Issues with Buttons*", in Proc. ACM Conference on Human Factors in Computing Systems CHI '90, Seattle, Washington, April 1990.

[Malone and Lai, 1992] Thomas Malone and Kum-Yew Lai, *"Experiments with Oval: A Radically Tailorable Tool for Cooperative Work"*, Proc. ACM Conference on Computer-Supported Cooperative Work CSCW '92, Toronto, Canada, Oct 31–Nov 4, 1992.

[Moon and Keene, 1986] David Moon and Sonya Keene, *"New Flavors"*, Proc. ACM Conf. Object-Oriented Programming, Systems, Languages and Applications OOPSLA '86, Portland, OR., September 1986

[Nardi and Miller, 1990] Bonnie Nardi and James Miller, *"An Ethnographic Study of Distributed Problem Solving in Spreadsheet Development"*, Proc. ACM Conference on Computer-Supported Cooperative Work CSCW '90, Los Angeles, October, 1990.

[Padget and Nuyens, *unpublished*] Julian Padget and Greg Nuyens (eds.), *"The EuLisp Programming Language Standard"*, unpublished draft.

[Rao, 1991] Ramana Rao,"*Implementational Reflection in Silica*", Proc. European Conference on Object-Oriented Programming ECOOP 91, Geneva, Switzerland, July 1991.

[des Rivières and Smith, 1984] Jim des Rivières and Brian Smith, "*The Implementation of Procedurally Reflective Languages*", Xerox PARC Technical Report ISL-4, Palo Alto, Ca., June 1984.

[Rodriguez, 1991] Luis Rodriguez, *"Coarse-Grained Parallelism Using Metaobject Protocols"*, Xerox PARC Technical Report SSL-91-06, Palo Alto, Ca., 1991.

[Smith, 1982] Brian Smith, "*Reflection and Semantics in a Procedural Language*", MIT Laboratory for Computer Science Report MIT-TR-272, 1982.

[Smith, 1984] Brian Smith, *"Reflection and Semantics in Lisp"*, Xerox PARC Technical Report ISL-5, Palo Alto, Ca., June 1984.

[Steele, 1984] Guy L. Steele Jr., "*Common Lisp: The Language*" (first edition), Digital Press, 1984.

[Trigg, 1992] Randy Trigg, *"Participatory Design meets the MOP: Accountability in the Design of Tailorable Computer Systems"*, Proc. 15th IRIS, in Bjerknes, Bratteteig and Kautz (Eds.), pp. 643–656, Larkollen, Norway, August, 1992.

[X3J13, 1988] Daniel Bobrow, Linda Demichiel, Richard Gabriel, Sonya Keene, Gregor Kiczales and David Moon, "*Common Lisp Object System Specification*", X3J13 Document 88-002R, June 1988.