System Design as Ongoing Principle

Mark Lycett  
*Brunel University*

Ray Paul  
*Brunel University*

Follow this and additional works at: [http://aisel.aisnet.org/amcis1998](http://aisel.aisnet.org/amcis1998)

**Recommended Citation**  
System Design as Ongoing Principle

Mark Lycett
Ray J. Paul
Department of Information Systems and Computing
Brunel University

Abstract

This paper proposes that traditional approaches to system development point to a conception of design that is flawed. One that leads to the design of static information systems that are then asked to work a dynamic world. This is articulated as the failure to treat design as an ongoing principle. In examining the case for ongoing design, component-based development is examined as a candidate approach.

Introduction

Since the articulation of the ‘software crisis’ (see Naur et al. 1976), a strong emphasis has been placed on the methodical approach to system development. Comprehensive reviews of the development process acknowledge this and provide evidence of an identifiable orthodoxy in information systems research in general (Hirschheim and Klein 1989; Hirschheim et al. 1995; Iivari 1991). The aim has been to introduce structure into the system development process by providing a purposeful framework that serves both to organise and regulate the activities of development and ensure that they are comprehensive. Whilst predominant, the methodical approach is by no means proven; as witnessed in the continuing search for an understanding of information systems failure (see Beynon-Davies 1995; DeLone and McLean 1992; Hirschheim and Newman 1988; Lyttinen and Hirschheim 1987; Poulmenakou and Holmes 1996; Sauer 1993). In addition, over the time that methodical approaches have evolved, environmental pressures now question the economics of large-scale system development (Baskerville et al. 1992; Grimes and Potel 1995).

Within this tradition, design acts as the link between the ‘capture’ of given aspects of human activity systems (organisational structures, practices and conventions for example) and their binary encapsulation in computer software. What is unfortunate is the constraint that methodical development places on design; namely that it requires design to be a contingent and predictive activity. It is our contention that this constraint leads to the development of static systems operating in a dynamic world. The paper begins by examining the nature of the world in which information systems are asked to work, alongside the way that methodical development forces design to treat this world. Our contention is that design should be treated as an ongoing process. With this made clear, the paper then examines an approach that offers some promise in such a direction, that of component-based development.

The Need for Ongoing Design

During the analysis and design stages of methodical development a variety of tools, techniques and methods of abstraction are used to model given aspects of human activity systems. At an abstract level, this approach is inherited from science, drawing on formalism and notions of optimality (Banbury 1987; Baskerville et al. 1992). A universal approach is sought to a class of problem situations, reducing them to an abstract set of symbols (often diagrammatic), thus allowing elements to be manipulated with a finite set of operations in order to deductively arrive at a ‘solution’. At a philosophical level, the methodical approach aims to assure that we develop credible knowledge about the present and possible future states of nature (Baskerville et al. 1992; Baskerville et al. 1996). These are embodied in representations, which typically deal with the structural, behavioural and functional aspects of a system (for example). The form of representation is partly constrained by the nature of the computer itself, which expresses complexity through a large number of physical switches and its ability to complete many simple binary operations in a short time. To this extent, the development of software represents a quest for greater abstraction in support of greater complexity (Laplante 1995).

Though this may be fine in isolation, it may be argued to be problematic in context. This is primarily because it fails to take account of ‘evolutionary complexity’ (see Lycett et al. 1997; Lycett and Paul 1998a; 1998b). If one accepts the complex nature of the social world, a constraint is introduced by the nature of the ‘concept to concept’ relationship inherent in the design process, where both the ‘observers knowledge’ and ‘knowledge of the observed’ may be placed under scrutiny. In terms of the abstraction process, there are two consequences of this that may naturally limit knowledge gained (Bhaskar 1979; 1989). Firstly, unacknowledged or unconscious motivation and tacit skills may limit stakeholders understanding of themselves during the process of analysis and design. Secondly, unintended consequences and unacknowledged conditions may limit the stakeholders understanding of the social world. These account for views expressed in the literature that ‘users cannot know what they want’ and ‘cannot explain what they know’ (see Ackoff 1967; Baskerville et al. 1992; Parnas and Clements 1986; Paul 1994).
addition, they also account for limitations in the analysis/design process, in part reflecting the argument that software descriptions that abstract away complexity often abstract away essence (Brooks 1987).

In dealing with future states of knowledge these problems are exacerbated. Evolutionary complexity proposes that social structures, mechanisms and processes (the entities that are abstracted) are emergent and thus may only be relatively enduring. Logically, this indicates that any ‘model’ may only be relatively enduring. By contrast, methodical development assumes social structures, mechanisms and processes are ‘invariant regularities’ that only have to be revealed to be understood (Baskerville et al. 1992). The process for revealing these regularities (embodied through design) is encapsulated within a project that, by nature, limits the time and resources allocated to development.

This combination engenders a dangerous philosophical habit that leads to the ‘reification of abstractions’ (see Jensen 1981). Developers abstract from a changing, evolving context and apply the abstraction as a fixed concrete property. Thus, the ability of the technical complexity of a system to satisfy the social complexity that it models (its internal variety) is implicitly designed in. The practical adequacy of the resultant model (as an acceptable representation of its real world counterpart) will critically depend on the degree of independence of ‘what was included in the model’ and ‘what was not’. In the short term such approaches may adequately represent complexity. In the long term it is argued that the resultant systems will always disappoint (or fail outright) as they do not have the means to evolve their internal variety in line with change in their environment. They represent temporal ‘snapshots’ that ultimately leave us with static systems that are asked to operate in a dynamic world.

The pragmatic of maintenance provides credibility for the argument to date. Maintenance consumes some 70 – 80 percent of total development cost (see Gilb 1988; Pressman 1992) and can consume up to 90 percent of intellectual effort (Giddings 1984). Crucially, some 88 – 98 percent of total maintenance cost is spent on enhancement, adapting the system to a changed environment (Fitzgerald 1990). This implicit realisation of evolutionary complexity, in contrast to explicit methodical ignorance, forces the designer to ‘future-proof’ where possible; primarily by attempting to predict or control change and/or ‘design in’ contingencies for change. For example, within the context of a development lifecycle, change analysis attempts to discern which requirements are most sensitive to change and which areas are most sensitive to changes in requirements (see Fitzgerald 1990; Strens and Sugden 1996). In a commercial context, many developers attempt to design software in a generic enough fashion to ‘please most of the people most of the time’. In this respect, the bloated nature of much modern ‘shrink-wrapped’ software is the lament of some notable authors (Wirth 1995; Yourdon 1996).

**Design as Ongoing Process**

In attempting to predict change and/or provide adequate flexibility to meet it, the difference between planning for the future and planning the future is confused (Paul 1994). The concentration of minds on the likelihood, possible effects and cost of change is laudable, but accounting for a series of ‘what-ifs’ is no guarantee of ‘what-will-be’. The lesson that we should learn from this is that design should be thought of as an *ongoing process*, not in a predictive or contingent sense. Approaches based upon the notion of ongoing design are rare (Olerup 1991).

As a first step in providing practical support for the principle of ongoing design, we propose the following:

1. The provision of a system architecture that allows a system some means to grow, adapt and evolve within a given environment.
2. The provision of mechanisms that allow a system dynamically to inherit or acquire modifications that make it better suited to a given environment.

With this in mind, we argue that the develop-from-scratch techniques that dominate development thinking should give way to techniques that emphasise construction from reusable building blocks (Garlan et al. 1995). We see this as a fruitful approach in attempting to design systems that are capable of dealing with evolutionary complexity.

**Component-Based Development**

Component-based development is not a new idea and can be traced at least as far back as the NATO sponsored Garmisch conference of 1968 (see McIlroy in Naur et al. 1976). The component ideal views the software element of an information system as a dynamic composition of reusable, pre-tested and independently upgradable components (Adler 1995; Cox 1990; Garlan et al. 1995; Murer 1997; Nierstrasz and Dami 1995; Szyperski 1998). Holistically, components are constituents – they represent ‘black-box’ units of independent production, acquisition and deployment that interact within the confines of an architecture to form a functioning system (Szyperski 1998). As units of independent production, components can be developed by different people, at different times, in complete ignorance of each other (Weck 1997). As units of acquisition and deployment, organisations benefit from reduced cost and risk associated with commercial-off-the-shelf software.

By definition, components are designed for composition (Nierstrasz and Dami 1995; Szyperski 1998). In ideal terms, a system designer selects the functionality they require, purchases the appropriate components and *composes* a system from them (Weck 1997). This conception is particularly pertinent as it infers that design is partitioned into a facet that is concerned with component design and a facet that is concerned with system design. The former does not preclude a methodical approach, though it does relegate it to ‘method in the small’. The latter is central to the realisation of the principle of ongoing design, as it provides the mechanism(s) for enabling a system to inherit or acquire modification that maintains the coupling between the system and
its environment - point (2) above. A key to this is the architectural approach in point (1) above. The resulting system will be both open and flexible if its software architecture is designed to be explicit and can be manipulated (Meijler and Nierstrasz 1998). For component-based architectures the vital factor is enabling a duality which provides both the basis for component independence and interoperability. This translates to a set of ‘policy decisions’ that curb variation by limiting the degrees of freedom given to mechanisms of communication, co-operation and co-ordination for example. A component architecture would thus likely consist of a set of platform decisions, a set of component frameworks and a design for the interoperability of those frameworks (Szyperski 1998). In turn, a component framework may be seen as a set of policy decisions that provides the common ground allowing multiple components to coexist in a single environment (Weck 1997). To this extent it represents a dedicated and focused architecture that provides (a) the protocols or contracts for interaction and (b) the dimensions for extensibility. Szyperski (1998) proposes that the key contribution of a component framework is the partial enforcement of architectural principles - in forcing components to perform certain tasks via mechanisms that are controlled by the framework, the framework retains the ability to enforce policies.

To date, no general model for software composition yet exists, though several research projects are attempting to address the perceived issues (Nierstrasz and Meijler 1995). Similarly, no current architectural approaches go beyond individual component frameworks (Szyperski 1998). The challenges here are not light, though some comparison may be drawn with the design of object-oriented frameworks whose difficulties are well noted (see Bäumer et al. 1997; Fayed and Schmidt 1997; Gamma et al. 1995; Garlan et al. 1995; Johnson 1997; Schmid 1997). That said, we propose that the potential for enabling the principle of ongoing design is clear.

**Conclusion**

In failing to account for evolutionary complexity, the methodical approach confuses ‘designing for the future’ with ‘designing the future’. The lesson that we should learn from this is that design should be thought of as an ongoing process, not a predictive or contingent one. In this respect, we have proposed that component-based development may bear some fruit. In separating the design of components from the design of component systems, the means and mechanisms for growth, adaptation and evolution are enabled. Components represent independent units of production, acquisition and distribution that are designed for reuse. Component systems represent contextual units of ongoing design enabled by an explicit architecture that can be manipulated. The dynamic mechanism allowing growth, adaptation and evolution is that of composition. This may be argued to provide a controlled approach for dealing with the challenge of evolutionary complexity.

We do not propose that such an approach represents a ‘silver bullet’ with regard to the principle of ongoing design; to do so would be both naïve and premature. Our only claim is that it offers significant promise. To fulfil such promise, important technical and social hurdles will have to be overcome. These cannot be divorced, for whilst it may be feasible to sustain imperfect technology in a working market, it is unlikely that perfect technology can be sustained where there is no market (Szyperski 1998). Our ongoing research currently seeks to analyse the promise of component-based development in practice. This research is undertaken in the context of a large software company currently seeking to develop component-based systems for the financial sector.

**References**

References are available on request from the first author upon request (Mark.Lycett@brunel.ac.uk; http://www.brunel.ac.uk/research/clist/).