Australasian Journal of Information Systems **Special Issue 2003/2004**

VERNACULARISM IN SOFTWARE DESIGN PRACTICE: DOES CRAFTSMANSHIP HAVE A PLACE IN SOFTWARE ENGINEERING?

Paul R. Taylor
(ptaylor@csse.monash.edu.au)
Agentis International, Level 2, 33 Lincoln Square South, Carlton, 3053.

ABSTRACT

Convention dictates that an information discipline matures from an informal shared practice to a defined and repeatable process through the externalisation and formal expression of its underlying theory. The inevitability of this progression should not be allowed to over-shadow the essential values, skills and knowledge transfer mechanisms of the superseded vernacular or craft-like practices. This paper examines the tension between software engineering's professionalisation of the software design role—exemplified by the software architect—and its antithesis, the software craftsperson, a characterisation that continues to emerge despite attempts to suppress reliance on individual skills and abilities through software engineering process. In other design disciplines, the professionalisation of design marks a distinct progression from ad hoc, unrepeatable, unselfconscious craft to a self-conscious, demarcated type of design found in most forms of engineering. Software engineering has partially failed to make this transition and this failure undermines the exclusivity of the engineering metaphor and engineering-based process models as a model of practical software design. Software methods must acknowledge and find ways of incorporating vernacularism and informality if the creative act of design is to be correctly characterised, supported in methods, and taught.

"Traditional vernaculars are themselves the great mass media; that is, specialized frames and vehicles of experience" — Marshall McLuhan.

DESIGN AND PROGRESS

An historian who took the time to examine the chronicles of software engineering could be excused for concluding that 'design' has always had the status of second class object. In software engineering, design finds expression in the techniques of first-class things such as programming languages, architectures, modelling techniques and the mechanical *modus operandi* of software production. We comprehend design and exercise our design knowledge and skills indirectly via the domain-specific filters of common software design techniques—user interface prototyping, pattern-driven architecture evolution, and use case analysis, for example. Unlike other disciplines in which design also plays a central role—architecture and industrial design being the two exemplars—software engineering has not yet elevated design beyond its expression in techniques to professional specialty, to a communication medium for the explicit reflection of local or societal values, or as a fulcrum for social or economic change. Questions about the relationships between individual practice, theory and collective professionalism are at the core of what it is we do as software designers.

Design in the software domain primarily serves the purpose of development and delivery, or of bringing product to market, and it must be presumed that the discipline, industry and community make this so. Identifying possible reasons for this necessitates both a specialised understanding of the nature of software and its design as well as a broad exposure to design beyond any one particular domain. The relative immaturity of software design at the system and architectural levels repeatedly observed. Shaw (1996), among others, claims that software architecture is an *ad hoc*, immature craft-like activity, applied only in an idiomatic fashion. *Ad hoc* design activity is generally discredited in software engineering, but not so in design theory outside software (Jencks & Silver, 1973), where the 'creative leap' of synthesis, or the enigmatic assembly skills of the bricoleur can explain outstanding acts of design. These 'designerly' behaviours have interesting implications for a perspective of design in general (Louridas, 1999), as well as software design (Taylor, 2000a).

The need to progress from craft to science is taken for granted in almost all applied sciences. This attitude may be seen to have its roots in the human struggle for ascendancy, the basis of science and the scientific method (Weatherall, 1979). Those who have questioned the need and appropriateness of this progression in design (such as Christopher Alexander in architecture) have at times been

branded brave, stupid or both (Grabow, 1983). In the software discipline, the need to professionalise design has clearly been motivated by economics—software development processes prone to bottlenecks at key design points did not scale and were therefore not feasible in the emerging software application marketplace of the 1970s, and the force of these economic drivers has scaled several orders of magnitude since. Whereas a craftsman designs and makes a one-off piece of furniture, jewellery or linen in isolation, the software designer's artefact is more the scale of a house, bridge or office tower and the complexity of a thousand such domestic or decorative objects. The analogies are rich but can equally be specious (Baragry & Reed, 2001).

The progression towards ever-increasing levels of design professionalism presents two important questions—is it the right progression in the case of software, and what are the implications of this progression for the ways we design software? The first question challenges the canon, even the *status quo*, with all of the attendant risks. An attempt at distilling the design consequences of an alternate metaphor, one that counters this notion of inevitable progress has been started (McBreen 2002)(Taylor, 2001). The remainder of this paper examines the second question—how has software design been shaped by the force of professionalisation through software's pursuit of engineering rigour, and how has the performance of software design changed as a result?

FROM VERNACULAR TO RATIONAL DESIGN

Design, both noun and verb, is so pervasive that it risks losing common meaning from its expression in so many objects and activities. Essentially, it refers to the idea at the root of a work (Hauffe, 1998). Software design is pervasive, creative, and multi-dimensional (Glass, 1999). It is widely regarded as a foundation of all systems development (Booch, 1994). Design problems of the class that are typically tackled by software projects have been referred to as 'wicked' problems (Budgen, 1994, Rittel & Weber, 1984), meaning they have non-linear, unpredictable and unmanageable characteristics. A predictor of how the role of design in software creation has and may still change is the historical trajectory of design through the industrial revolution, which saw vernacular design practices replaced by the rational models and approaches that now underpin engineering and the architecture of the built world.

SELF-CONSCIOUS AND UNSELFCONSCIOUS DESIGN PROCESSES

Some observers detect this progression in the designer's self-awareness. Alexander (1964) introduced a broad distinction between two kinds of design processes, one he called 'unselfconscious', the other 'self-conscious'. Roughly speaking, the unselfconscious process is that which goes on in primitive societies or in the traditional craft or architectural vernacular contexts, while the self-conscious process is that which is typical of present-day, educated, specialised professional designers and architects.

The way that the design and the production of useful objects is taught provides the best differentiator. In the unselfconscious case, the teaching of craft skills is through demonstration and imitation of skilled craftsman. Thus the novice learns by practising the actual skill. In the 'self-conscious' process, the techniques are taught by being formulated explicitly and explained theoretically. In the unselfconscious culture, the same form is repeated over and over again, and the individual craftsperson must learn how to copy a prototype. But in self-conscious design there are always new contexts of use arising for which traditional solutions are inappropriate or inadequate. These necessitate some degree of theoretical understanding in order to be able to devise new forms to meet the new needs.

In software creation, the unselfconscious designer—perhaps working in isolation to merge experience with skill, evolving the function, structure and aesthetics of a complex software architecture or component in piecemeal fashion—is a believable, if not suppressed image. A fundamental question is whether we as methodologists and design theorists believe that a model or paradigm should *dictate*, or *reflect* practice. A more manageable question is which model—or what overlay of models—best describes how software design is known to be performed.

THE VALUE OF VERNACULARISM

The techniques, tricks, mores and means passed from building to building via builders are informal, causal and astute. Design as a communication medium—or the ability of a building or artefact to harbour representations of its designer's intentions beyond basic function—has long been recognised by architectural theorists (Eco, 1997), and vernacular design does this with maximum efficiency. 'Vernacular' is a term borrowed from linguists by architectural historians to mean 'the language of a region'. Vernacular buildings are those not designed by architects or professional designers. Familiar examples include the Cotswold villages, Cape Cod houses, and any other style that evolved locally under the collective influences of isolation, climate, locally available materials and cultural stability. In architectural terms, vernacular buildings are seen as the opposite of whatever is contemporary, high style, or polite. Dormer (1988) writes that "the craft aesthetic is anti-technology, anti-science and anti-progress" (p. 135). It is little wonder that, in their clamour for engineering credibility, early software methodologists rejected vernacularism outright.

Vernacularism, however, has much to recommend it. Vernacular buildings evolve extremely well. As each successive generation of design imitates its predecessor, the best buildings, techniques and uses of materials are copied while the less ideal are forgotten. According to Brand's (1994) analysis of how buildings 'learn' (that is, how they evolve to embody fit-for-purpose design over time), the heart of vernacular design is about form, not style. The evolutionary development of a vernacular style influences rooms and rooflines rather than colours or stylistic appliqué. The most striking feature of Brand's vernacularism is the effect of human use and habitation over time as the predominant selector of designs—"style is time's fool; form is time's student" (p. 133). Others have written on theoretical aspects of the evolution of designs (Hull, 1988, Steadman, 1979, Stebbins, 1971) and design knowledge (Gero, 1996, Kaplan, 2000). Some related work focuses on evolution in software design (Foote & Opdyke, 1995, Kemerer & Slaughter, 1999, Lieberherr & Xiao, 1993, Taylor, 2000b).

Vernacularism appears to be robust and irrepressible. It is wrong to assume that the professionalisation of design expunges all traces of vernacularism—rather, it exists to a degree in all design contexts. Even as the most prominent and extreme Modernists attempted to rid early twentieth century design of its human foibles by seeking a new kind of beauty in the ultimate elevation of function over form and aesthetics, a kind of 'industrial vernacular' was created (Berman, 1988). The semiotic language of Modernist design was sparse and minimalist but a distinct, readable language nonetheless, with touches of regional colour and recognisable personal idiosyncrasies (Venturi et al., 1977).

Some software design theorists regard vernacular design as the only kind that has actually been seen to work. Borenstein (1991), for example, thinks that anecdotes (programmer's stories, word of mouth, narratives and case studies) convey more software design knowledge than any theory or method. A primary motivation of the software design patterns movement is to express common software designs that are known to have occurred in at least three different contexts (Coplien, 1996). The so-called Agile methods and eXtreme Programming promote less formal approaches to design and knowledge transfer over 'hard' theoretical ones.

USING ROLES TO ILLUSTRATE THE TRANSITION FROM VERNACULAR TO RATIONAL DESIGN

If vernacularism cannot be eliminated in the pursuit of rationalism, it must exert its influence by degrees. A litmus test for the presence or absence of vernacularism in a design context is the relationships between stakeholders — the designer, the maker and the user. Mayall (1979) views these relationships between the pre-industrial designer or craftsman and the user as a simple closed loop—the craftsman designs and makes for the patron directly. Its vital characteristic for healthy

design is unhindered communication and a tightly bound, responsive feedback loop. This simple closed loop relationship still exists between designers and craftspersons and their patrons today. It also occurs between many architects and their clients, and consultants and their clients. It is so pervasive because it works so well. Its simplicity leads to some obvious problems—it does not scale well, the customer may not know what to request or may interfere too closely with the designer's work causing constant re-direction; or the customer may get too involved in the design process or otherwise not comprehend the represented complexity. Walker and Cross (1976) elaborate on the vernacular 'mode of design' in Figure 1.

These diagrams are read as follows: the rectangles with rounded corners represent roles, the product is shown as a circle, and dialogue is represented as a triangle. The exertion of pressures and needs (i.e. requirements) is represented by a jagged line superimposed over the thick arrow that represents the normal interactions during design and construction activity. In Walker and Cross' vernacular design (a), user, client, designer and maker are one, and the craftsperson is unimpeded in her control of the object. This is a representation of classic craftsmanship, and if you substitute 'object' or 'component' for 'product', it is equally applicable to the software developer's routine work of detailed design, coding, testing, and evolving a software artefact. If the object has no visibility to other developers or to the software system's user, and is a consequence of the software developer's particular design, the illusion is complete, and the cameo of the software craftsperson emerges. When the designer designs a product for someone else (or when the developer iterates the development of a component that provides user interface or services to a system user) the designer is still free to use vernacular solutions and techniques but must select, synthesise and arrange them to meet the negotiated requirements of the patron (b).

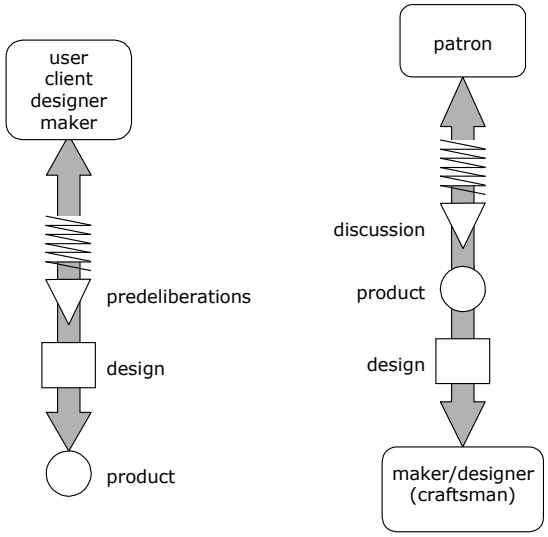

Figure 1: Two representations of vernacular design; (a) vernacular design; (b) the empirical exchange (Walker & Cross, 1976, p. 58).

THE SEPARATION OF DESIGNING AND MAKING

The craftsman's direct experience of materials, patterns of use and clients is in sharp contrast with the isolation of post-industrial product designers from the manufacture, use and users of their designs. Software methodologists may legitimately blame the inability of vernacular design to scale, or to be managed in contemporary business environments, as the primary reasons for its rejection. But in industrial design it was the emergence of new materials in standard sizes, forms and quantities that motivated the advent of designer as separate from manufacturer.

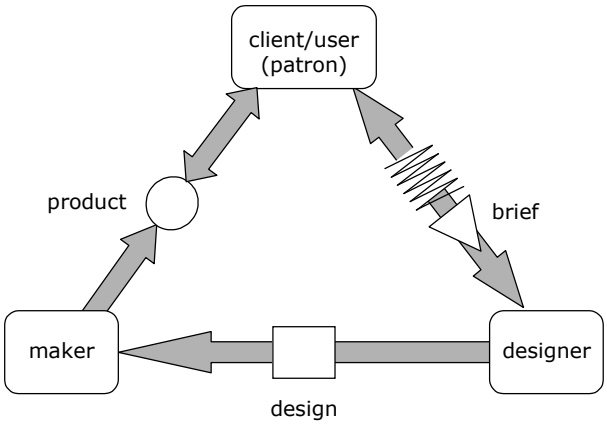

Figure 2: Direct patronage (Walker & Cross, 1976, p. 57).

In Walker and Cross' depiction (Figure 2) a new token appears—a rectangle that represents 'the design' in a form that the maker can interpret. The design is the designer's interpretation and transformation of the brief from the client, user and/or patron. The fact that the patron and user interact only with the designer and the product, and never with the maker, reinforces the commodification of making in this (post-industrial) paradigm. The separation of designing and making opened the door for professionalisation, as typified by the architecture profession. Now, one group laid claim to the high ground of the conceptual and aesthetic dimensions of design whilst delegating the detailed engineering and structural design to others. This division of responsibilities split concept from detail, image from functionality, and momentary perception from long-term occupation. The emergence of rational design processes sacrificed responsiveness and contextual relevance for uniformity and universality. Each additional identification and separation of roles in time and space brought additional interfaces, and greater reliance on specification and process. The success of professionalised design in modern architecture and industrial manufacturing appeared as a glimmering chimera to those in other design domains looking for proven models to adopt.

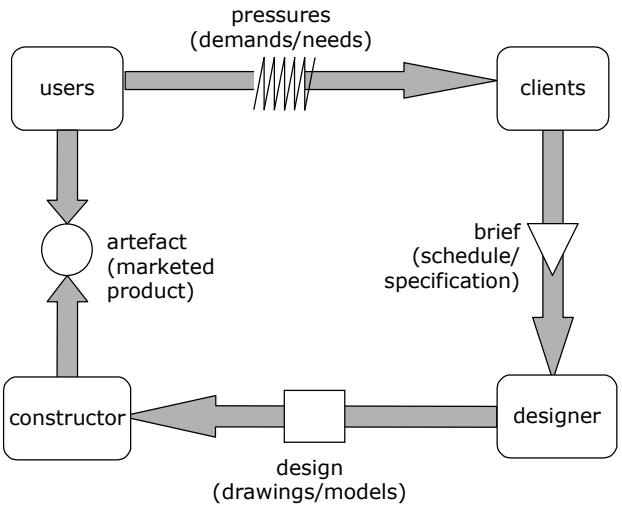

Figure 3: The rational network (Walker & Cross, 1976, p. 56).

RATIONAL DESIGN

The success of the separation of designing and making in industrial manufacturing should not be

interpreted as a universal justification for the elimination of vernacular design in software development—rationalism in design brings costs and consequences. Mayall's description of the emergence of rational design emphasises the loss of feedback loops. According to Mayall, machine-based industrialisation caused the break-up of the designer/craftsperson into the four roles. The management–design–manufacturing split was a direct consequence of the growth of specialisation stimulated by the industrial revolution and larger, international markets that depended upon high-volume manufacture. The breakdown of communication between designer and user followed from the introduction of new types of product and manufacturing processes that could not be judged or assessed until they had been tried out for some time. Feedback loops became strikingly elongated. Walker's depiction of rational design completes the picture that we are so familiar with today (Figure 3). Users are depicted as a separate stakeholder group from the designer's clients. Ultimately, users create the forces that motivate the design and construction processes, and users evaluate the results. The three major artefacts of the rational design process are the brief (requirements), the design (drawings and models), and the artefact or product itself. Most contemporary software methodologies assume this model of the context and function of design, and these roles and relationships dictate the methodological framework by which software artefacts are produced (Truex 2000).

PROBLEMS WITH THE RATIONAL DESIGN PARADIGM

Historically, experience with the rational design paradigm suggests some consistent themes. These include the disconnection of design from manufacture, the disconnection of design from the context of use, the elongation of critical feedback loops, and the repression of 'natural' informal and vernacular design practices.

A good place to start a critique of rational, post-industrial design is with the claim that rational design actually works on realistic problems—that it delivers workable solutions. Jean Claude Garcias, writing on France's changing cities (Myerson, 1993) observed that the myth underlying the thinking of Le Corbusier and his colleagues was that of zoning—that by carefully re-ordering the city, by separating its functions, it would be possible to solve social ills. But functional decomposition, or master-planning as it is known outside of software, has not worked absolutely, and many observers now claim that Modernist design principles found in many of the world's modern cities left a legacy of structural and habitation problems (Koestler 1998). The rational master-plan failed to recognise that the most vibrant, stimulating and functionally coherent cities are those in which the city's structure allows for localised control and expression, and that these properties are emergent rather than consequential. This assertion has been made about other designed structures, including software structures (Coplien, 2000a, Foote & Yoder, 1999).

The rational designer must work hard to avoid over-commitment to *a priori* master-planning. In designing complex structures embedded in larger, non-deterministic social systems, a solution structure can only designed for the context as it is perceived. This reality of rational 'snapshot-design' ameliorates the fiction of successful rational design for all eventualities within a given planning horizon. Inevitably, in software and in other structured systems, today's solution becomes tomorrow's problem.

The obvious antidote to one-shot design is evolutionary or emergent design process, but the rational paradigm is diametrically opposed to evolutionary design. During the analysis and design phases of a rational system development process, aspects of social complexity are modelled using a variety of tools, techniques and methods of abstraction. This methodological approach, often sold in the form of packaged product-methodologies and encased within a project culture, is inherited from science, drawing on formalism and notions of optimality. Scientific positivism impels the rational methodologist toward universal approaches to a wide family of problem situations, reducing them to an abstract set of symbols (often diagrammatic), thereby allowing a series of semi-autonomous transformations to deliver a solution. At a philosophical level, this approach aims to assure that we develop credible knowledge about the present and possible future states of the world (Lycett & Paul, 1998)—it is as much about effecting risk management as it is about predictable, repeatable design.

The methodological approach to systems development (Lycett argues) assumes social structures, mechanisms and processes as 'invariant regularities' that only have to be revealed to be understood. In the long term, methodological systems will—in most important cases—disappoint, as they do not allow internal variety to evolve in line with the environment. They are prone to represent temporal snapshots, ultimately leaving us with static systems that must operate in a dynamic world. Lycett concludes that design should be an inherently ongoing process, not one where the designer is required to predict or control change and articulate every requirement of a design. Continuous design was also recognised by design theorists outside of software, where a transition from a focus on product to process was advocated some time ago (Thackara 1986) (Jones, 1988, Mitchell, 1988).

In the software engineering literature, a growing band of authors have been questioning the exclusivity of the rational basis of the predominant metaphors and methods. Blum (1996) systematically deconstructs the current software engineering paradigm using the failures of positivism and universal science to jump the gap to the social sciences, and the situatedness of observed contextual design as evidence, and argues for an adaptive paradigm of software design. Truex (2000) reaches similar conclusions after applying his deconstruction of the 'privileged text' of IS development methods. Gabriel's Feyerabend project (2001) meets regularly to enact exercises and workshops to similarly deconstruct preconceptions of software construction. Baragry (2001) has claimed that software must abandon its associations with inappropriate metaphors (architecture and engineering) in order to be able to define a more appropriate philosophical basis. Many outside of software (Feyerabend, 1993) have expressed similar views.

Those of us working in the theories and paradigms of software design are at risk of repeating design history through ignorance. As a reaction against industrialisation's suffocation of design quality and craftsmanship, the Arts and Crafts movement originated in Great Britain and spread to America. Artisans who worked in the Arts and Crafts style attempted to revive the medieval guild system, re-establishing high quality standards of workmanship, and instilling the idea of 'truth to materials' as the basis of design (Naylor, 1990). Recent writings in software methods witness a similar reaction. Coplien (2000b) draws a direct parallel between the Arts and Crafts reaction and the replacement of large-scale industrial software processes (such as the Capability Maturity Model) by developer-centric lightweight methods. Beck's (2000) 'eXtreme programming' and SCRUM (Beedle et al., 2000) are methods based on pair programming and time-boxing that attempt to relocate design responsibilities on all levels of abstraction back to the individual developer. These 'agile' methods attempt to empower the individual developer within a larger scale project management methodology by separating project management and risk aversion responsibilities from design responsibilities. This difficult balancing act illustrates the essence of the critical challenge facing software methodologists—to find ways of combining rational and vernacular design modes within contemporary software design contexts.

IN DEFENCE OF THE CRAFT METAPHOR FOR SOFTWARE CREATION

Craft, as a metaphor, illustrates the process of making. It is concerned with the non-industrial acts of continuous design, creation, fabrication and evaluation. Craftsmanship implies highly developed skill, deep knowledge of the fabric and material, and an intimate integration of the craftsman's knowledge and expertise with the crafted object. Craftsmanship holistically integrates designing, making and using, and equally intermeshes conceptual and detailed design with all aspects of fabrication. The performance of this kind of making involves transparently and continuously moving between modes of designing, making, evaluating and reflecting, being totally immersed in and consumed by the work, using tacit and explicit skills and knowledge, often in ways that result in knowledge becoming embedded within the artefact.

The public looks to craft for perfection, mastery, and evidence of rare innate skill. Clear criteria and a widely, almost universally held appreciation of what makes up good work define craft. In stone-craft of all kinds the craftsmanship is appreciated universally across time (today's stonemasons learn from Greek and Roman ruins). The basis of this common understanding is the material itself—because stone, for instance, has never changed its basic building characteristics, examples of its use

for building are evident across the centuries. Wherever the material has the capacity to act as a medium of communication that transcends language, culture and time, a craft exists. Craftsmen collect around crafts-worthy materials—those that embody characteristics that reflect skill and attention to detail back to the worker and are malleable and pleasurable to work with. On the face of it, craft has much in common with the personal software development processes and practices used by experienced developers during intensive software development and evolution.

How does the software engineering literature reflect upon this inherent informality? Most interpretations are shaped by the positivistic roots of the applied sciences, and in any doctrine based upon functionalism the only alternative is to regard it as a shameful lack of control (Hauffe 1998). Clearly, pragmatic informality must be countered with the discipline of methodology. But the elimination of informality inherent in Modernist architecture lead to a form of authoritarian 'design control' that is evident to a degree in most software methods.

On the face of it, Dormer's earlier comment (that '*the craft aesthetic is anti-technology, anti-science and anti-progress*') seems like an irreversible demolition of the metaphor as it might apply to software development. But if we parallel Dormer's crafts and industries with the state of software industry, it can be argued that the metaphor gains strength rather than losing it. Software craft can only be called anti-technology, anti-science and anti-progress when the software industry approaches the level of determinism, automation, and maturity as the manufacturing industry has over the last century. By most comparisons, the software industry's current ability to 'manufacture' software goods places it somewhere in the analogical vicinity of the manufacturing industry's deployment of steam. In the light of this comparison, the craft ethic, aesthetic and metaphor in the software industry have relevance.

Professionalism—A Third Force on Design Practice

Figure **4** illustrates a view of the sources of the competing forces that this discussion has surveyed. Those forces that shape software design practice that do not explicitly fit rational method or vernacular craft performance are represented as 'professionalism'. The diagram of the left summarises current—albeit idealised—attitudes to rationalism, professionalism and vernacularism in software design thinking. This paper has argued that vernacular design is a profoundly natural and in many ways suitable perspective from which to think about design in the software fabric, but that the incumbent rational paradigm of software design has no place for vernacularism. Addressing this oversight creates the intersection that appears on the right of
Figure **4**, which raises important questions about which characteristics of software design belong in each of the set intersections, and why.

The influence of professionalism is a force not to be underestimated in shaping how the rational and vernacular elements of design might be brought together. A shared sense of professionalism filters practices and legitimates some techniques over others. But the emergence of techniques within the context of a standard body of knowledge is different to the professionalisation of design. Professionalisation selects models and modes of designing over time, and is subject to market pull and economic push. Professionalism draws selectively from rationalism, using rational arguments to support behaviours that suit the individual or the enterprise. It groups a set of largely socially constructed phenomena. The intersection of rationalism and professionalism must be carefully scrutinised in every software design landscape to discern which rational methods and techniques are appropriate and contribute genuine value and which are there for other reasons.

Professionalism-meets-vernacularism is an under-recognised and under-rated juncture. This intersection is typified by the use of the professional or collegiate network, reflection, altruism, mentoring, and enculturation. Here, designers seek out peers to source and validate potential solutions and experience for personal reasons. Software engineering methodology has not acknowledged the value of this behaviour, and as a result, some informal, apparently *ad hoc* designer behaviours are unnecessarily under-valued.

Rationalism-meets-vernacularism is a non sequitur in design theory and history. This intersection, to the degree that it exists at all, is best enumerated by countless accounts of the domination of rational design over vernacular craft. As noted, reactionary software design methods have begun to address this.

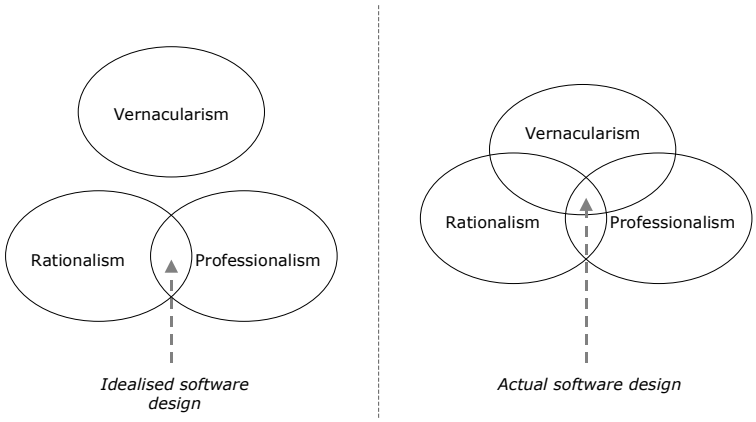

Figure 4: Representation of idealised and actual software design.

The intersection of professionalism, rationalism and vernacularism represents the greatest challenge to the applied field of software creation. No one would sensibly claim that rationalism in contemporary software design is fatally flawed. However, as has been argued, many observers of industrialisation, both within and outside of software, perceive problems in a unilateral devotion to rational philosophy. As the discipline of software engineering has attempted to transform software craft into a fully-fledged member of the engineering family, the role of software designer (amongst several others) has evolved. Careful observation of its changing features serves to illustrate the ever-present dangers of over-correction. The formalising forces of professionalisation of the craft of software development have had similar effect. Of interest to software engineering professionals and methodologists is the degree to which these problems reflect on treatments of design in contemporary methodology and what, if any action should be taken to mitigate the associated risks.

CONCLUSION

In summary, a number of objections to rational design have been raised. These include the disconnection of design from manufacture (a distinction that makes little sense in software construction); the disconnection of design from the context of use (a problem that has achieved more prominence in information systems literature than in software engineering); elongation of critical feedback loops, and repression of 'natural' informal and vernacular design practices. The problems that surface as a result of these mismatches stem from the underlying tension between the incumbent rational model of software design and other paradigms of thought and action that influence how practitioners approach software construction. This tension has the potential to retard progressive thought and innovative software design practice.

Marshall McLuhan's insightful prediction from 1970 has been demonstrated repeatedly by the pioneer software creators at Microsoft, Sun, Novell and countless others:

> "In terms of, say, a computer technology we are headed for cottage economics, where the most important industrial activities can be carried on in any individual little shack anywhere on the globe" — Marshall McLuhan.

Suggestions for paths forward range from incremental methodological evolution (as much of the current software engineering research would seem to suggest) to full-scale Kuhnian paradigm shift. Either way, the environmental factors that will select the methodological variants that prosper in the next chapter of the history of software creation—the global economy's insatiable demand for

software; the need for dramatic increases in quality, responsiveness and flexibility to name a few—will significantly test our current notions of rationalism, vernacularism and professionalism.

REFERENCES

Alexander, C. (1964) **Notes on the Synthesis of Form** (New York, Harvard University Press).

Baragry, J. & Reed, K. (2001) Why we need a Different View of Software Architecture**Working IEEE/IFIP Conference on Software Architecture** (Amsterdam, The Netherlands,

Beck, K. (2000) **Embracing Change: Extreme Programming Explained** (Cambridge, Cambridge University Press).

Beedle, M., Devos, M., Sharon, Y., Schwaber, K. & Sutherland, J. (2000) SCRUM: A Pattern Language for Hyperproductive Software Development, in: N. Harrison, B. Foote & H. Rohnert (Eds) **Pattern Languages of Program Design 4**, Vol. 4 (Reading, Massachusetts, Addison-Wesley).

Berman, M. (1988) The Experience of Modernity, in: J. Thackara (Ed) **Design After Modernism** (London, Thames and Hudson).

Blum, B. (1996) **Beyond Programming: To a New Era of Design** Oxford University Press).

Booch, G. (1994) **Object-Oriented Analysis and Design with Applications** (Redwood City, California, Benjamin Cummings).

Borenstein, N.S. (1991) **Programming as if People Mattered: Friendly Programs, Software Engineering and Other Noble Delusions** (Princeton, New Jersey, Princeton University Press).

Brand, S. (1994) **How Buildings Learn: What Happens to Them after they're Built** (New York, Penguin).

Brooks, F.P. (1987) No silver bullet: Essence and accidents of software engineering, **IEEE Computer**, 20(4), pp. 10-19.

Budgen, D. (1994) **Software Design** Addison-Wesley).

Coplien, J.O. (1996) Software Patterns(New York, Lucent Technologies, Bell Labs Innovations).

Coplien, J.O. (2000a) Architecture as Metaphor

Coplien, J.O. (2000b) Patterns and Art, **C++ Report**, 12(1), pp. 41-43.

Dormer, P. (1988) The Ideal World of Vermeer's Little Lacemaker, in: J. Thackara (Ed) **Design After Modernism** (London, Thames and Hudson).

Eco, U. (1997) Function and Sign: the Semiotics of Architecture, in: N. Leach (Ed) **Rethinking Architecture: A reader in cultural theory** (London, Routledge).

Feyerabend, P. (1993) **Against Method** (London, Verso).

Foote, B. & Opdyke, W.F. (1995) Lifecycle and Refactoring Patterns that Support Evolution and Reuse, in: J.O. Coplien & D.C. Scmidt (Eds) **Pattern Languages of Program Design** Addison-Wesley).

Foote, B. & Yoder, J. (1999) Big Ball of Mud, in: N. Harrison (Ed) **Pattern Languages of Program Design 4**, Vol. 4 Addison-Wesley).

Gabriel, R.P. (2001) The Feyerabend Project (http://www.dreamsongs.com/FeyerabendW1.html)

Gero, J.S. (1996) Creativity, emergence and evolution in design: concepts and framework, **Knowledge Based Systems**, 9(7), pp. 435-448.

Glass, R.L. (1999) On Design, **IEEE Software**, Mar/Apr 1999, pp. 104-103.

Grabow, S. (1983) **Christopher Alexander: The Search for a New Paradigm in Architecture** (Chicago, University of Chicago Press).

Hauffe, T. (1998) **Design: A Concise History** (London, Laurence King Publishing).

Hull, D.L. (1988) **Science as a Process: An Evolutionary Account of the Social and Conceptual Development of Science** (Chicago, The University of Chicago Press).

Jencks, C. & Silver, N. (1973) **Adhocism: The Case for Improvisation** (New York, Anchor Books).

Jones, J.C. (1988) Softecnica, in: J. Thackara (Ed) **Design After Modernism** (London, Thames and Hudson).

Kaplan, S.M. (2000) Co-Evolution in Socio-Technical Systems**Computer Supported Cooperative Work 2000** (Philadelphia, ACM).

Kemerer, C.F. & Slaughter, S. (1999) An Empirical Approach to Studying Software Evolution, **IEEE Transactions on Software Engineering**, 25(4), pp. 493-509.

Lieberherr, K.J. & Xiao, C. (1993) Object-Oriented Software Evolution, **IEEE Transactions on Software Engineering**, 19(4), pp. 313-343.

Louridas, P. (1999) Design as bricolage: anthropology meets design thinking, **Design Studies**, 20(6), pp. 517-535.

Lycett, M. & Paul, R.J. (1998) Information Systems Development: The Challenge of Evolutionary Complexity in: W.R.J. Baets (Ed) **Sixth European Conference on Information Systems** (Aix-en-Provence, France, Euro-Arab management School, Granada, Spain.).

Mayall, W.H. (1979) **Principles in Design** (New York, Van Nostrand Rienhold).

McBreen, P. (2002) **Software Craftsmanship: The New Imperative** (Addison Wesley Professional).

Mitchell, T. (1988) The Product as Illusion, in: J. Thackara (Ed) **Design After Modernism** (London, Thames and Hudson).

Myerson, J. (1993) Design renaissance: Selected papers from the International Design Congress in: J. Myerson (Ed) **International Design Congress** (Glasgow, Scotland, Open Eye Publishing).

Naylor, G. (1990) **The Arts and Craft Movement: A Study of its Sources, Ideals and Influence on Design Theory** (London, Trefoil Publications).

Rittel, H.J. & Weber, M.M. (1984) Planning problems are wicked problems, in: N. Cross (Ed) **Developments in Design Methodology** Wiley).

Shaw, M. & Garlan, F. (1996) **Software Architecture: Perspectives on an Emerging Discipline**

Steadman, P. (1979) **The Evolution of Designs: Biological Analogy in Architecture and the Applied Arts** (Cambridge, Cambridge University Press).

Stebbins, G.L. (1971) **Processes of Organic Evolution** (Englewood Cliffs, New Jersey, Prentice-Hall, Inc.).

Taylor, P. (2000a) Adhocism in Software Architecture - Perspectives from Design Theory in: J.G.a.P. Croll (Ed) **International Conference of Software Methods and Tools (2000)** (Wollongong, IEEE Computer Press).

Taylor, P. (2000b) Evolution of Software Design Knowledge in: F. Burnstein (Ed) **Australian Conference on Knowledge Management and Intelligent Decision Support (2000)** (Melbourne, Monash University).

Taylor, P.R. (2001) Patterns of Software Craft in: J. Noble & N. Harrison (Eds) **Second Australian Conference of Pattern Languages and Programs (KoalaPLoP 2001)** (Melbourne, University of Wellington).

Thackara, J. (1988) Beyond the Object in Design, in: J. Thackara (Ed) **Design After Modernism** (London, Thames and Hudson).

Venturi, R., Scott Brown, D. & Izenour, S. (1977) **Learning from Las Vegas** (Cambridge, Massachusetts, The MIT Press).

Walker, D. & Cross, N. (1976) **Design: The man-made object** The Open University Press).

Weatherall, M. (1979) **Scientific Method** (London, The English Universities Press Ltd.).

Dormer, P. (1988). The Ideal World of Vermeer's Little Lacemaker. **Design After Modernism**. J. Thackara. London, Thames and Hudson.

Hauffe, T. (1998). **Design: A Concise History**. London, Laurence King Publishing.

Koestler, H. (1998). **Home from Nowhere**.

McBreen, P. (2002). **Software Craftsmanship: The New Imperative**. Boston, Addison-Wesley.

McIlroy, D. (1969). **Mass Produced Software Components**. First NATO Conference on Software Engineering.

Thackara, J. (1986). **New British Design**. London, Thames and Hudson.

Truex, D. (2000). Amethodical Systems Methodology.

Venturi, R., D. Scott Brown, et al. (1977). **Learning from Las Vegas**. Cambridge, Massachusetts, The MIT Press.