

Postmodern Prospects for Conceptual Modelling

James Noble

Computer Science
Victoria University of Wellington
New Zealand
kjx@mcs.vuw.ac.nz

Robert Biddle

Human-Oriented Technology Lab
Carleton University
Canada
robert.biddle@carleton.ca

Abstract

A number of recent developments in software engineering — from agile methods to aspect-oriented programming to design patterns to good enough software — share a number of common attributes. These developments avoid a unifying theme or plan, focus on negotiation between different concerns, and exhibit a high level of context sensitivity. We argue that these developments are evidence of a *postmodern turn* in software engineering. In this paper, we survey a number of these developments and describe their potential implications for the practice of conceptual modelling.

Keywords: Postmodernism, Conceptual Modelling

1 Introduction

A spectre is haunting software development — the spectre of *Postmodernism!* A number of recent developments in software engineering — from agile methods to aspect-oriented programming to design patterns to good enough software — share a number of common attributes. These developments avoid a unifying theme or plan, focus on negotiation between different concerns, and exhibit a high level of context sensitivity.

In this paper, we will argue that these developments are evidence of a *postmodern turn* in software engineering. The notion that software development could usefully be analysed as postmodern was first identified by Hugh Robinson and his colleagues around ten years ago (Robinson, Hall, Hovenden & Rachel 1998). In their prophetic article *Postmodern Software Development* they write:

We have seen that the development methods of software have been grounded in modernism and the Enlightenment, but that this has led to conflicts and contradictions, and a notion of ‘software crisis’ as the modernist metanarrative has broken down.

The views expressed here are that the postmodern ethos can offer some emancipation to the process of software development. In questioning the accepted rules and values and, crucially, not offering any others in their place, software development has to become a more locally negotiated phenomenon. The rules you follow are the ones that are suggested by the situation at hand.

Copyright ©2006, Australian Computer Society, Inc. This paper appeared at Third Asia-Pacific Conference on Conceptual Modelling (APCCM2006), Hobart, Australia. Conferences in Research and Practice in Information Technology, Vol. 53. Markus Stumptner, Sven Hartmann and Yasushi Kiyoki, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

Robinson’s article is primarily concerned with the process of software development. More recently, in a series of *Notes on Postmodern Programming* (Noble & Biddle 2002, Biddle, Martin & Noble 2003, Noble & Biddle 2004) we have considered more pragmatic artifacts of software development, programming and programming technology, including languages and design, and the construction of systems from existing software.

In this paper, we draw on this work, plus some crucial theories of the postmodern, to raise questions of postmodernism for conceptual modelling. This paper is in three parts: in the first part, we describe some of the developments and classifications of postmodernism as they have developed in other disciplines, outside software development. Then, in the second part, we will survey a number of developments within software engineering and describe how they exhibit postmodern features. Finally, we conclude by proposing some potential prospects for conceptual modelling faced with the spectre of this postmodern turn. Of necessity, this paper is a survey, rather than an in depth treatment, and to avoid disappointment, be aware that we aim to raise questions, rather than provide answers.

2 Postmodernism

The first question that is generally raised when discussing *postmodernism* is simply: what does postmodernism mean?¹ If modern means “now” or “today”, how can something happen that is *after* what is now or what is today? Indeed, this is a good question: one of the first postmodern theorists, Jean-François Lyotard, describes postmodernism as “*the paradox of the future (post) anterior (modo)*” — that is, the postmodern is what comes after the future (Lyotard 1992, Lyotard 1984, Wikipedia 2005).

While a love of paradox, in-jokes, and irreverence is certainly a typically postmodern style, for the purpose of pedagogy, in this paper we can adopt a more prosaic (but equally unhelpful) definition, derived from the *Notes on Postmodern Programming* (Noble & Biddle 2002, Biddle et al. 2003, Noble & Biddle 2004):

*postmodernism applies modern means
to other ends*

In the case of software development, computer science, software engineering, then, postmodernism often means applying (or misapplying) the fruits of the disciplines’ development over the last fifty years, but applying them a way, or for a purpose, that their original developers would oppose.

¹The traditional answer to this question is that the margin of this paper is insufficient to contain a definition (Noble & Biddle 2002).

In the remainder of this section, we will explore the definition of postmodernism in general; proceeding to considering postmodernisms in software development in the next.

2.1 Incredulity towards Meta-Narratives

In the foreword to his seminal text *The Postmodern Condition*, (Lyotard 1979, Lyotard 1984) the philosopher Jean-François Lyotard characterised postmodernism as “*incredulity towards meta-narratives*”. At little more length, Lyotard writes:

Le recours aux grands récits est exclu; on ne saurait donc recourir ni à la dialectique de l'Esprit ni même à l'émancipation de l'humanité comme validation du discours scientifique postmoderne. Mais, on vient de le voir, le “petit récit” reste la forme par excellence que prend l'invention imaginative, et tout d'abord dans la science.

Recourse to grand narratives is forbidden; we cannot resort to the dialectical of the Spirit nor even to the emancipation of humanity for validation of postmodern scientific discourse. But, as we have seen, the “small narrative” is a form which superbly allows imaginative invention, and most of all in science.

In brief, what Lyotard means is that the sustaining common myths of (Western) civilisation (Christianity, Marxism, Newtonian Physics, Progress) no longer have the strength that they held prior to the twentieth century. There is no longer one *big story* to which all of society may subscribe. To present a brief example, the Somme then Auschwitz illustrate that progress, mechanisation, and industrialisation are not themselves necessarily a universal good, the therefore progress is not to be sought as an end in itself. Similarly, the lack of churchgoing in most Western countries, the adoption by most Western Labour parties of private-sector solutions to social problems, and the vicissitudes of relativity (not to mention string theory) illustrate that other grand narratives (Christianity, Socialism, Newtonian Mechanics) are no longer normative. More pragmatically, perhaps, the developments of software technologies and the communication technologies they undergird have resulted in a world where almost everybody is just a phone call away, or virtually present in our living-rooms via television and the Internet. And as recent events continue to show, there is no shared grand narrative by which we may live.

We argue that computer science similarly has had a grand metanarrative: a big story that we are careful to teach first-year students and hold up as the idea of the systems we design and build. In computer science, the primary goals are that software must be correct and efficient — software engineers often add in that the software must be maintainable and usable. The first two introductory concepts of the 2001 ACM Computing Curriculum are *algorithmic computation* (effectively correctness) and *algorithmic efficiency* (The Joint Task Force on Computing Curricula 2001). As Martin Rinard has pithily described, software that fails to meet these goals is seen as evidence of a *moral* weakness on behalf of its programmers (Rinard, Cadar & Nguyen 2005a).

We argue that software development now comes under the postmodern condition: efficiency and correctness, the grand narratives of the discipline, are no longer an effective or useful guide to practice.

2.2 Negotiation and Context Sensitivity

The death of meta-narratives raises a practical problem: without an overarching framework, how can one make decisions? In computer science, if we should not be aiming to build correct software, or efficient software, then what should be doing? Indeed, this is another major critique of postmodernism — that without a metanarrative, all one is left with is uncritical relativism — that “*anything goes*”.

This is a caricature of Lyotard's position, however. The absence of a single, overriding grand narrative does not mean that decisions may be taken without reference to any external referent. Rather, in place of a single privileged master narrative, we must contend with many localised small narratives, and reconcile them in making a decision. Thus, rather than simply following a grand narrative — Marxist theory, Christian ethics, or in software, a methodology like Structured Design, Stepwise-Refinement, or the Unified Method — we need to determine what small narratives are “in play” at any particular time, and then attempt to resolve the conflict between them. To quote Lyotard (1984):

... the principle that any consensus on the rules defining a game, and the “moves” playable within it must be local, in other words, agreed on by its present players and subject to eventual cancellation.

In practical design or modelling activities, this negotiation between many small narratives typically manifests itself as sensitivity to the context of a model, design, or decision. In architecture, for example, a tenet of postmodern design is sensitivity to the actual location, to the physical site where the building will be constructed. This is in contrast to modern architecture, where (modern) technology dominates all other considerations, and so you build a large rectangular concrete-steel-and-glass box wherever you happen to be (Jencks 1987).

In software design and engineering, for example, postmodern development is more likely to be concerned with existing software products, the values of development and client organisations, the expertise and experience of the development staff, rather than seek to impose an overarching methodology.

2.3 Double Coding

Negotiation between many small stories, and then fitting a design into a particular context (or contexts) leads to one of the key characteristics of postmodern designed *artifacts* — in software in as much as architecture, product design, music, or other art forms. This is that postmodern designs typically use *double coding* (or *multiple coding*) to appeal to more than one audience or meet more than one concern.

The easiest examples to see of this are in architecture: rectangular concrete-and-steel-and-glass boxes are notoriously uncongenial places in which to work and live, not to mention being indistinguishable from each other. Postmodern architects, then, using the same technologies as their modern forebears to actually construct buildings, will add on decorations, spires, domes, arcades, and other ornaments both to distinguish builds from each other, and also to humanise their buildings as homes or workplaces (Jencks 1987).

We find similar multiple codings in software. Contemporary desktop operating systems (Windows, Linux, Macintosh) are good examples here: just as a postmodern architect may paint a mural (or at least

colour the building materials) on the side of a building, add a fancy three-dimensional dome over the doorway, and use contrasting materials for visual effect on walls or floors, so personal computers and mobile phones add fancy screen backgrounds, customised sound effects and ringtones that does not change the functionality of the underlying software but presumably makes it more appealing to its users.

Double coding gives rise to another common criticism of postmodernism — that postmodernism prefers surface to depth, appearance to reality, lying to truth, or cute cross references (Sanrio Company, Ltd. & Sanrio, Inc 2005) to straightforward explanation. And indeed, this criticism is valid to a point (Waugh 1943): one (or more) of the codes are often pastiches of existing or historical styles that are laid over other foundations or applied to other purposes.

Some well-known examples of double-coding in software are the Java programming language and the latest Macintosh operating systems. Java was sold as a successor to C++ — adopting much C++ syntax and terminology. In terms of its underlying technology, Java is effectively a Smalltalk system with types on the bytecodes and an overcomplicated compiler — and indeed, advanced Java programming techniques, relying on garbage collection, finalisation, reflection, dynamic code generation, and so on, are much more closely related to Smalltalk practice than to C++ practice. Similarly, the current (and near future) Macintosh laptops — X86 architecture, open source Darwin kernel, KHTML-based web browser, etc — is almost exactly the same as current Linux or BSD-based systems apart from the details of the design of the window system, and the supremely important logo on the back of every screen.

Double coding and constant negotiation often give rise to a sense of pervasive irony — parody, pastiche, or knowingness is a characteristic of much postmodern writing, architecture, and design. The *Hacker's Dictionary* contains many examples (Raymond & Steele 1993); more recent ones include the pervasive use of cartoons to illustrate computer science textbooks, a programming languages named after a coffee shop, and digital rights management software that rootkits your computer with illegally copied code. This irony is now pervasive: we have operating systems called “Vista”, and positioned with skyscrapers: but we also have operating systems with a fat penguin as a logo, that promote stability as its main advantage.

2.4 Reuse

Finally, where modernism sets itself against past designs and past practice, postmodernism treats the past as just another set of small narratives. Where modern computer science chose to invent a discipline from nothing, a postmodern approach can incorporate all the techniques and experiences from the sixty years the discipline has existed — holding them as contingent, to use as needed.

In some ways, this is evidence of the success of software: we do have a history of successful techniques that have been used to build successful systems. Furthermore, in the last ten to fifteen years, we have also constructed a range of successful software components and subsystems that can be incorporated into new systems. Especially in industrial practice, most development involves existing systems — either for so-called “maintenance” (most often adapting well-functioning successful systems to incorporate new requirements), or as larger-scale developments which graft new functionality and technologies onto existing, long-lived systems, or indeed in some cases replacing systems wholesale.

This has a number of effects in practice. One is that mastery of a programming language, modelling notation, or algorithm or data structure implementations is not enough to equip a professional programmer: rather, programmers and designers also need knowledge of existing systems — sensitivity to the technical context of the development — and of available libraries, components, and frameworks that they can combine into programs.

2.5 Postmodern Responses

In different disciplines and in different contexts, postmodernism responds in different ways to forces described above. Across these disciplines, however, we can generally identify a number of stereotyped responses to postmodernity (or equally different types of postmodernism, or different postmodernisms). The architect and architectural critic Charles Jencks has identified four kinds of architectural postmodernisms. Based upon his classification (Jencks 1987), in this section we outline what we consider the four key software responses to postmodernism: neoclassicism, eclecticism, antimodernism, hypermodernism.

2.5.1 Neoclassicism

Especially within architecture, the most common postmodernism is a variety of *neoclassicism* — also known as postmodern classicism. In postmodern architecture, neoclassicism adopts the standard construction techniques from modern architecture, but adds features drawn from classical architecture to “humanise” the buildings, and to provide a consistent “skin” over the building.

Much contemporary software and language design is neoclassical — simply as a way to smuggle (post) modern technologies into practice. Java or C# are two good examples here: their syntax is carefully designed to look like C or C++, yet their semantics are much closer to Lisp or Smalltalk. This shift is obvious in Apple’s Dylan language, where the first version used Lisp-style S-expression syntax (Shalit 1992), while the second version adopted something much close to C or Pascal (Shalit, Moon & Starbuck 1996).

2.5.2 Eclecticism

The other main postmodern response is *eclecticism* which throws together a number of different features or components, resulting in a collage or melange of textures or styles, with no apparent common thread. IM Pei’s glass pyramid in front of the Louvre is one example: Frank Gehry’s Stata Centre for MIT’s CSAIL lab is another. This response, drawing heavily on quotations or allusions, is common to postmodern music and visual arts, as well as architecture.

A number of postmodern software systems and languages exhibit eclectic postmodernism. The most well-known is the Perl programming language. Perl’s designer, Larry Wall, has explicitly described Perl is the “first postmodern programming language” and discussed how he explicitly and indiscriminately borrowed features from many other programming languages (both “high culture” languages such as Pascal and BCPL, and “low culture” languages such as BASIC or TRAC) to produce Perl’s design. Still within programming languages, two complementary examples are the recent text *Higher-Order Perl* (Dominus 2005) (with a back cover blurb promising to teach Perl programmers “powerful programming methods — new to most Perl programmers — that were previously the domain of computer scientists”), and use of Haskell to build DOM Components (Finne, Leijen, Meijer & Jones 1999).

Our previous work, the *Notes* (Biddle et al. 2003), is written in an eclectic form: this paper is neoclassical (it at least *looks* like a scientific paper). The advantage of eclecticism is that — by simply combining different ideas, components, or styles, it doesn't "paper over" significant difficulties or differences between them: a neoclassical approach, as in this paper, can appear to tell a straightforward, consistent story about something that is neither straightforward nor consistent.

2.5.3 Antimodernism

The third postmodern response, which we call *anti-modernism*, is less common in software development, but is more common both in pedagogy and also commercial analysis and modelling practices. This is an essentially low-technology approach, and loosely parallels the "ecological" approach that Jencks identifies in postmodern architecture (Jencks 1987).

An ironic icon for this approach could be Dijkstra, in abandoning the typewriter or word processor and hand-writing all his EWD letters². In pedagogy, techniques such as *Computer Science Unplugged* (Bell, Fellows & Witten 1998) attempt to teach the basics of computer science to schoolchildren or university students without a computer, while Ronald Stamper has advocated *Informatics Without the Computer* as an approach to both systems analysis and an attempt to broaden systems analysis techniques to be applicable to non-computer systems (Stamper 2001). And computer science seems full of curmudgeonly statements of form "X is an improvement over all its successors".

More practically, low-technology approaches have been advocated as tools, especially for analysis or modelling. Class-Responsibility-Cards (CRC Cards), although originally proposed as a learning tool, are accepted as an effective group-based analysis tool (Beck & Cunningham 1989); Coplien has advocated the use of a whiteboard rather than CASE tools (with a separate "mercenary analyst" role to move information into CASE tools if necessary) (Coplien & Harrison 2004); the design patterns movement has resisted automation or case tool support for patterns (Coplien 1996).

Examples of antimodernism in *systems* design seem to be rather harder to come by — presumably because an antimodern approach would be to eliminate the computer system, and such decisions are not often described in the literature of the discipline. The "Wiki Way" is one successful example of antimodern design (Leuf & Cunningham 2001), allowing users to create web sites by using minimal textual markup within any web browser, rather than complex integrated authoring environments.

2.5.4 Hypermodernism

The last response we will consider is the continuation, or the exaggeration of modern design and aesthetics. One way to think about this is that eclectic postmodernism can coöpt any past or present style: one of those it can coöpt is modernism. Then, in spite of what we argue is the general failure of modern master-narratives of software development, research and development can still continue working directly from modern premises.

The hypermodern response most practically manifests itself as a kind of minimalism: building systems based on a single language or single architecture, which may be correct, very powerful, and very

²Luca Cardelli provides a ironic, eclectic overlay on this refusal, with a "handwritten" postscript printer font called "Dijkstra" (Cardelli 2005)

efficient, but because of a modernist rejection of existing practice, are unable to be directly adopted or used widely. The extensible, customizable self-documenting Lisp-based emacs editor is one such system (Stallman 1981); in spite of all its power and elegance of design, it has never been widely adopted outside a circle of cognoscenti.

The key difference between this extreme postmodern response, and the modern grand narrative, is that, say, while Wirth was developing Pascal could do so with the idea that an independent, completely-Pascal-only system could well become widely used — as indeed it did. Twenty years later, few people were willing to install a new operating system, learn a novel user interface paradigm, and then a new programming language to use Oberon.

3 The Postmodern Turn

Having toured postmodernism, in this section we will turn to software, and consider how some recent developments may be infected with postmodernism. This selection is not intended to be complete or consistent. We have certainly omitted many important developments (Open Source (Raymond 2001), UML, XML, various kinds of multiparadigm programming, customisable methodologies), some of which are discussed elsewhere in the *Notes*.

3.1 Agile Methods

The first postmodern development we consider here is the introduction of so-called Agile software development methods. The most well-known (and arguably most adopted) Agile methodology is Extreme Programming (Beck 1999). The Agile software movement has the advantage of a manifesto, show in Figure 1 (The Agile Alliance 2001):

In postmodern terms, traditional software development projects stands under a grand narrative of "progress". Expert analysts "extract" or "mine" requirements from clients in roughly same spirit oil and coal are extracted or mined; once the experts have sufficient knowledge of the terrain, they can the proceed to manufacturing the software. Often this manufacturing is carried out in roughly the same techniques as industrial production lines and the end of the process, the final software is shipped to clients and the process is over (Robinson et al. 1998).

Strange as it may seem, the idea of an Agile systems development is that it does not make any progress. Agile development treats all development as maintenance programming, keeping an existing system running, and evolving it through a series of iterations, not as a greenfield development from nothing. Ideally, at the end of any Agile iteration, a system will suit its operational context (functional requirements) as well as its development context (principally resources) allow: the system does not grow more complete or more complex — or even necessarily larger over time (especially as features and their supporting code may be removed once they are no longer required).

In practice, of course, Agile developments do have a model of progress — the number of user stories or functions completed, the amount of "backlog" in a Lean or Scrum development (Poppendieck & Poppendieck 2003, Schwaber & Beedle 2001). But this progress is temporal, limited, and local: as a whole, the system and its developers exist in a steady, *stable* state (Beck 1999).

Thus the key point of the agile manifesto — rather than applying a manufacturing methodology, customers and developers are seen as bringing their own

Manifesto for Agile Software Development

We are uncovering better ways of developing software
by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Figure 1: Agile Manifesto

knowledge about their own spheres of expertise, (their own little narratives) and they continually negotiate changes to the extant working software to be completed in the next iteration. Even though traditional methodologies may build software iteratively, the overall scope and cost of the construction effort is generally estimated and then established in advance. In contrast, Agile developments prefer variable scope contracts, perhaps with a fixed cost-per-unit-time, during which the software will be maintained or evolved. To maintain this focus on the actual software and personal interaction with clients, many Agile methodologies, in particular XP, downplay or actively forbid any attempts to maintain software models, plans, or analysis or design documents. Perhaps whiteboards and notepaper may be permitted, but they must be erased or disposed of at the end of each working day. Ownership of domain knowledge remains with the customer representatives; the structure of the software is only represented by its source code.

In terms of our classification of postmodernisms above, Agile methods at a large scale are antimodernism — eschewing complex modern estimation and planning methods, analysis and design tools and methodologies in favour of direct communication and (ephemeral) paper based aids similar to CRC Cards. At small scales, Agile methods promote adoption of advanced code analysis and programming tools, such as refactoring browsers, that can automatically analyse and reshape code to incorporate new requirements, and unit testing regression to ensure these changes preserve the code's behaviour: in this sense, they are neoclassical. On the other hand, some Agile methods (again, particularly XP) try to promote their own (hypermodernist) grand narratives, advocating strict adherence to all the practices to be considered “truly” practicing XP (Portland Patterns Repository 2005). In our research into actual Agile development teams (Martin, Biddle & Noble 2004a, Martin, Biddle & Noble 2004b), however, we have found that most development teams in practice take a rather more flexible approach, adopting those practices that are perceived to fit within their development context, and ignoring those that do not.

3.2 Aspect-Orientation: Software Development without Knowledge

Traditional software development effectively enjoys a unified ontology: software has single hierarchical program structure, a tree (Dahl & Hoare 1972). Depending on the type of program, this may be based either on a functional decomposition (as in structured programs); part-whole aggregation hierarchies (common in entity-relationship modelling and non-

relational databases) or classification hierarchies (especially object-oriented modelling). We consider that this structure represents a grand narrative: a program is a single unitary artifact, a consistent encoding of domain knowledge gained by a modeller and created by a programmer.

Aspect-Oriented Software Development (AOSD, also known as Advanced Separation of Concerns) is based upon a fundamental critique of this structure for programs (Kiczales, Lamping, Mendhekar, Maeda, Lopes, Loingtier & Irwin. 1997, Harrison & Ossher 1993). AOSD points out that programs in fact consist of multiple interlocking issues or concerns, so that each individual single program element in a traditional tree structure (say a class definition) will *tangle* code related to a number of different concerns (data storage, database access, user interfaces) while any single concern will be *distributed* across many different program elements. AOSD claims this tangling and distribution causes a range of problems with program comprehension, maintenance, and construction, and as a result, proposes that program structures should be multifaced and multilinked, rather than tree shaped. Programming language constructs need to be revised to support these new structures, so that programs can break down programs' structure into many small components, each corresponding to an individual concern, and then somehow combining those parts back together into a functioning program.

In fact, AOSD's critique is not new, harking back to Fred Brooks' description of the “*intricately interlocking software elephant*” (Brooks 1987). Neither is its solution new: the phrase: “*separation of concerns*” has a long history within computer science (Dijkstra 1982, Parnas & Clements 1986). The difference in AOSD is that this decomposition is heterarchical, rather than hierarchical: a graph, rather than a tree³. Each separate component or separate concern plays the role of a small narrative in the program's design: modelling or programming languages explicitly have to embody strategies to negotiate between this local narratives (such as composition rules in subject-oriented programming, or aspect dominance in Aspect/J).

This kind of cross-cutting, interlinked structure is common to many different postmodern critiques, including antimodernism architect Christopher Alexander's description that “*A city is not a tree*” (Alexander 1965), or eclectic postmodern philosophers Deleuze and Guattari's notion of a rhizome (Deleuze & Guattari 1987):

³Observant readers will realise that the *implicit* structure of any program is a (multi)graph, and modern, structured languages restrict only the *explicit* structure of a program to be a tree. AOSD proponents argue this is precisely the point: the aim of AOSD is to make explicit as much of the structure of the program as possible.

The rhizome itself assumes very diverse forms, from ramified surface extension in all directions to concretion into bulbs and tubers. When rats swarm over each other. The rhizome includes the best and the worst: potato and couchgrass, or the weed. Animal and plant, couchgrass is crabgrass.

AOSD programs in current practice can be considered neoclassical, as typically they consist of “base” programs in a modern programming language with a tree topology, but with extra aspects being weaved in to handle one or to extraordinary functions, such as program tracing, monitoring, debugging, concurrent synchronisation, or security. More recent developments ranging from formal models (Aldrich 2005) to modelling all interobject relationships as aspects (Pearce & Noble 2006), lean more towards hypermodernism, capturing as much of a programs’ implicitly interwoven structure explicitly in aspects.

3.3 Design Patterns

Existing methodologies are also based on grand narratives of design, describing how to construct entire programs by developing requirements top down into a design then into consistent code. This is as true for traditional structured methodologies (Yourdon & Constantine 1979), object-oriented methodologies (Wirfs-Brock, Wilkerson & Wiener 1990), or even aspect-oriented methodologies (Clarke & Baniassad 2005). In that sense, methodologies embody an ethic of what is “good” over whole programs, and so design all of a program or system in the same way.

In contrast, design patterns (Gamma, Helm, Johnson & Vlissides 1994) are little local rules about what is good in a design, or rather, that describe and evaluate a characteristic, reusable design in such a way that programmers can apply this design to their programs when it is needed. The classic definition of a pattern, “*A solution to a problem in a context*” both encapsulates the specificity of a pattern (one solution to one problem) and also its context dependence (Coplien 1996, Gamma et al. 1994). Because patterns are small, specific, and contingent, they can be adopted by programmers without changing their existing methodologies or work methods, as and when required, irrespective of what other design discipline has been adopted. Patterns may be employed singularly, or in combination with either each other, or with other techniques or programming constructs or techniques. As John Vlissides describes in his *Pattern Hatching* (Vlissides 1998):

You will discover techniques for applying certain patterns and not applying others, as circumstances dictate.

In terms of the typology of postmodern responses above, design patterns are generally an eclectic postmodernism, as multiple patterns from multiple sources may be used indiscriminately together in a program — indeed, we consider this is a key reason for the patterns movement’s success. Parts of the patterns movement are antimodernist, in that, like Agile development, they eschew automated support (or indeed any kind of external analysis) of patterns (Coplien 1996). Refactoring, which emerged from the same milieu as patterns, is not antimodernist in this way. On the other hand, also like some XP and AOP proponents, those patterns proponents who follow Christopher Alexander (the originator of design patterns in Architecture (Alexander 1979, Alexander 1977)) are simultaneously hypermodernist and antimodernist, on one hand, insisting that all patterns

must be joined together in a grand, overarching *pattern language* — another grand narrative — and on the other, dismissing academic analysis or automated support for program design

Patterns are also related to a number of other programming techniques that have similar effects. Refactorings (Opdyke & Johnson 1990, Fowler 1999) are generally smaller than patterns, describing localised code changes that improve (or at least modify) the structure of a program. Analysis Patterns (Fowler 1997) and Problem Frames (Jackson 2001) are again descriptions of particular localised, partial solutions but addressed earlier in the traditional software lifecycle.

3.4 Good Enough Software

We discussed above that key parts of the Computer Science grand narrative are the twin goals of correctness and then efficiency: these are non-negotiable — although correctness is more important. From this perspective, the aim of developing only “*good enough software*” — software that is neither correct nor efficient — but is good enough for its context of use, is also a postmodern approach.

Gabriel’s essay *Lisp: Good News, Bad News, How to Win Big* (Gabriel 1991) includes an early elucidation of this argument, arguing that a design that emphasised simplicity and sacrificed completeness would have better survival characteristics than a design that promoted correctness and completeness.

Agile approaches, such as XP, take this reasoning further: accepting that all design qualities — correctness, efficiency, completeness, consistency, simplicity, development time, and cost — are independent variables in a development process (Beck 1999). A particular development project needs to negotiate between all these small narratives of software characteristics to fit the projects’ context, perhaps trading off quality for price, for development time, for the project scope (for a simple website development) or perhaps delivering correctness at any cost (for a life-critical system). Rinard’s Acceptability-Oriented Computing (Rinard, Cadar & Nguyen 2005b) goes one step further, arguing that the programming and design techniques programmers typically adopt to ensure programs’ correctness actually reduce the correctness and reliability of the systems of which those programs are parts.

The concept of good enough software is an antimodernist response to development pressure: rejecting the modern claims that software should be correct and efficient. The complementary contrary aim of developing perfect software through the use of formal methods is the hypermodern antithesis to this. Perhaps there is also a neoclassical synthesis, where tools like Findbugs (Hovemeyer & Pugh 2004) or Spec# (Barnett, Leino & Schulte 2004) employ sophisticated formal analyses to catch potential errors, but do not guarantee the correctness of the resulting program.

3.5 Scrap-Heap Programming

Steven Conner (Connor 2004) describes postmodernism as

...that condition in which, for the first time, and as a result of technologies which allow the large-scale storage, access, and reproduction of records of the past, the past appears to be included in the present.

Nowhere is this aspect of postmodernism more explicit than in software development. This is for two reasons: first, digital technologies are the key enablers of this large-scale storage, access, and reproduction;

and second, that software itself is the content *par excellence* that can be stored and exchanged using these technologies. In effect, everything is becoming software.

This shift has a major impact on software development. The context of a software development projects now includes huge amounts of existing software, available via Google on the the web, provided by the open source movement, purchased one way or another from software vendors — and, often most pertinent — pre-existing software within the client organisation which the new software must extend, cooperate, or replace. To self-plagiarise: “*How do you write a program when every program has already been written*” (Noble & Biddle 2004).

Our techniques for modelling and programming must take account of this existing software — and in particular, of the changes it makes to the structure of the software that we build. To return again to our theme: once software is constructed mostly out of other software, there is no longer any one grand narrative, big story, of software structure: no longer any one coherent software design. Rather, we build software in a scrapheap (Noble & Biddle 2002, Moore & Pryce 2005, Brucker-Cohen & Moriwaki 2005, Channel 4 2004). The end result is the “*Programmer as DJ*” — (re)mixing existing systems, scavenging parts of systems, buying or building discrete components and then writing or generating amorphous glue that binds these disparate parts together.

The modern grand narratives of program design generally work from requirements to design to engineering. This model holds good for both Agile and non-Agile methodologies: in fact, Agile methodologies such as XP often have an even stronger focus on this separation of labour: a customer provides requirements while the programmer implements them (Beck 1999). Much postmodern software engineering tends to work bottom up — not bottom up from the requirements, but bottom up from the scrapheap. Programmers begin with the components that they have to hand, or can find on Google, or can scavenge; then work out what they could build out of the components; and only then negotiate with customers over requirements.

Actual assembly of scrapheap software is often not via programming in traditional programming languages, but rather via scripting languages, languages that are good enough for the job. Traditional programming languages are designed for writing whole programs: scripting languages are designed for writing small parts of them: the interstitial filler between the parts Google delivers from the scrapheap.

These little languages are low culture languages — shell scripts, Perl, Ruby, Visual Basic — generally ignored by academic computer scientists, software engineers, or modellers. The languages are interpreted and dynamically typed, good enough in that they are concerned neither with correctness (offered by static typing) or efficiency (compilation) — but are nevertheless powerful and efficient enough to write full applications, given that they make it very easy to connect together other software. Where these languages are discussed (Bently 1988), they are generally not studied in their own right, but rather as implementation techniques to build a new customised language for a particular problem. While building a new language is a scrapheap technique, the most common case is to simply use one or more of these languages to glue existing components together.

Brian Foote has captured the Scrapheap dynamic as the difference between a *Big Ball of Mud* (Foote & Yoder 2000) and a *Big Bucket of Glue* (Foote 2005). Foote describes LISP as a Big Ball of Mud — a very flexible syntax, language, system, to which program-

mers can always add more Lisp code because it always remains just a Big Ball of Mud. But the important thing about a Big Bucket of Glue is not the glue: the glue is what sticks together the stuff pulled from the scrapheap. Two buckets of glue can be very different, depending upon what scraps they contain. Programs are composites of multiple things, a conglomerate rather than a mixture, different parts, stuck together with glue that is different again from any of the components: mud is undifferentiated.

So we have composed software, assembled software, software bricolage, (Biddle et al. 2003) — software made out of a range of parts of different sizes, languages, platforms, and technologies. These parts may not necessarily have been designed to be encapsulated reusable components: they will have (implicit) dependencies on architecture, standards, other software that may only be partially understood. Building programs out of existing software results in radically heterogenous systems — because the parts are not just simple “components” (Cox 1990) but may be whole other applications, web services, entire complex legacy interlinked computer systems accessed via screen-scrapers, facades, or wrappers of various sorts. The cost of merely understanding these software, let alone reengineering them to fit some overarching software architecture, would be prohibitive.

Scrapheap programming is in some ways the most obviously characteristic postmodern programming style. In terms of the postmodernisms above, Scrapheap programming is eclectic, highly context sensitive, mixes high and low culture (using anything that will do the job, from Haskell to Visual Basic) and heavily double coded (facades, wrappers, user interfaces are ubiquitous to make scrapheap software seem like something it is not).

4 Prospects for Postmodern Conceptual Modelling

The rubric for the First Asia-Pacific Conference on Conceptual Modelling (Hartmann & Roddick 2004) states:

The actual amount, complexity and diversity of information are increasing day by day. Information that has to be conceptualised and efficiently organised in order to be useful. Conceptual modelling is fundamental to the development of up-to-date information and knowledge-based systems.

What, then, are the prospects for a postmodern approach to conceptual modelling? First, it seems that this rubric already accepts one of the tenets of postmodernism — the increasing volume, complexity, and diversity of information, and perhaps an increasing anxiety to keep our systems always “up-to-date” with a postmodernity that is always just after the future we imagined when we first designed the systems. Second, though, the statement stands on a modern assertion of metanarrative: information has to be *conceptualised and efficiently organised*, in order to be useful. Arguably, Google provides a counterexample: information can be useful given only moderately efficient access, without correctness (conceptualisation) or efficient organisation, and certainly without a grand narrative to hold everything together.

Although, in postmodernity, there is likely no privileged “high ground” that conceptual modelling can claim, this does not mean it has no place in postmodern software development. Rather, conceptual modelling can negotiate as a little narrative, along with many other techniques for the “*development of up-to-date information and knowledge-based systems*” —

with Visual Basic and category theory and Microsoft Bob and abstract interpretation and CRC cards and Haskell.

But, how could such negotiation work in practice? How do you practice conceptual modelling for Visual Basic programs, or for an Agile methodology that explicitly rejects modelling and design? Thankfully, the prospects for aspect-oriented conceptual modelling seem a little brighter, given conceptual modelling's existing move from its straightforward entity-relationship heritage to encompass some object-oriented techniques, and more recently towards XML. How can conceptual modelling adapt to projects where other narratives (cost or time to market) outweigh the traditional values of correctness and efficiency? Ultimately, what utility is there in conceptual modelling a system built piecemeal out of junk found from a scrapheap via Google?

Perhaps it is more useful, as we have done to some extent here, to consider how a particular technology or technique responds to postmodernity, rather than simply asking whether or not something is "postmodern" (especially where "postmodern" is often meant as a synonym for "bad"). If our analysis is correct, then we are all working in postmodernity: an anti-modern response — clinging heroically, desperately, to one ideal imagined future where all information is conceptualised and all databases are normalised — is as much a postmodern response as gleefully claiming eclecticism removes the need (or responsibility) for developers or modellers to make any value decisions. Rather, the point is that every decision is now a local negotiation between various contingent narratives. So, as researchers, we can ask "how is this postmodern?", or "what aspects of this are postmodern?", or "what kind of postmodern response is this?". How is conceptual modelling sensitive to context? What kind of double or multiple codings are in effect? How can conceptual modelling, aware of its status as one little narrative amongst many, contribute to the ongoing negotiation at the core of systems development in the postmodern era?

References

- Aldrich, J. (2005), Open modules: Modular reasoning about advice, in 'ECOOP Proceedings'.
- Alexander, C. (1965), 'A city is not a tree', *DESIGN* pp. 46–55.
- Alexander, C. (1977), *A Pattern Language*, Oxford University Press.
- Alexander, C. (1979), *The Timeless Way of Building*, Oxford University Press.
- Barnett, M., Leino, K. R. M. & Schulte, W. (2004), The Spec# programming system: An overview, in G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet & T. Muntean, eds, 'Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)', Vol. 3362 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 49–69.
- Beck, K. (1999), *Extreme Programming Explained: Embrace Change*, Addison-Wesley.
- Beck, K. & Cunningham, W. (1989), A laboratory for teaching object oriented thinking, in 'OOPSLA Proceedings'.
- Bell, T., Fellows, M. & Witten, I. (1998), *Computer Science Unplugged: The Original Activities Book*, Computer Science Unplugged, Morrisville, NC. www.unplugged.canterbury.ac.nz.
- Bently, J., ed. (1988), *More Programming Pearls*, Addison-Wesley.
- Biddle, R., Martin, A. & Noble, J. (2003), 'No name: Just notes on software reuse', *SigPlan Notices: Proceedings of the Oopsla Onward Track* **38**(2), 76–96.
- Brooks, Jr., F. P. (1987), 'No silver bullet: Essence and accidents of software engineering', *IEEE Computer* **20**(4).
- Brucker-Cohen, J. & Moriwaki, K. (2005), 'Scrapyard challenge workshop', <http://www.-scrapyardchallenge.com/>.
- Cardelli, L. (2005), 'Dijkstra postscript font', <http://www.luca.demon.co.uk/Fonts.htm>.
- Channel 4 (2004), 'Scrapheap challenge', <http://www.channel4.com/science/microsites/S/-scrapheap/>.
- Clarke, S. & Baniassad, E. (2005), *Aspect-Oriented Analysis and Design: The Theme Approach*, Addison-Wesley.
- Connor, S., ed. (2004), *The Cambridge Companion to Postmodernism*, Cambridge University Press.
- Coplien, J. O. (1996), *Software Patterns*, SIGS Management Briefings, SIGS Press.
- Coplien, J. O. & Harrison, N. B. (2004), *Organizational Patterns of Agile Software Development*, Prentice Hall PTR.
- Cox, B. J. (1990), 'Planning the software industrial revolution', *IEEE Software*.
- Dahl, O.-J. & Hoare, C. A. R. (1972), Hierarchical program structures, in O.-J. Dahl, E. W. Dijkstra & C. A. R. Hoare, eds, 'Structured Programming', Academic Press.
- Deleuze, G. & Guattari, F. (1987), *A Thousand Plateaus: Capitalism and Schizophrenia*, University of Minnesota Press. Translated from the French by Brian Massumi.
- Dijkstra, E. W. (1982), On the role of scientific thought, in 'Selected Writings on Computing: A Personal Perspective', Springer-Verlag, pp. 60–66.
- Dominus, M. J. (2005), *Higher-Order Perl*, Elsevier Science & Technology Books.
- Finne, S., Leijen, D., Meijer, E. & Jones, S. P. (1999), Calling hell from heaven and heaven from hell, in 'Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP-99)', ACM, pp. 114–125.
- Foote, B. (2005), Big bucket of glue (breakthrough idea), in 'OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications', ACM Press, pp. 76–86.
- Foote, B. & Yoder, J. (2000), Big ball of mud, in N. Harrison, B. Foote & H. Rohnert, eds, 'Pattern Languages of Program Design', Vol. 4, Addison-Wesley, chapter 29, pp. 653–692.
- Fowler, M. (1997), *Analysis Patterns*, Addison-Wesley.
- Fowler, M. (1999), *Refactoring: Improving the Design of Existing Code*, Addison-Wesley.

- Gabriel, R. P. (1991), 'LISP: Good news, bad news, how to win big', *AI Expert* **6**(6), 30–39.
- Gamma, E., Helm, R., Johnson, R. E. & Vlissides, J. (1994), *Design Patterns*, Addison-Wesley.
- Harrison, W. & Ossher, H. (1993), Subject-oriented programming (a critique of pure objects), in 'OOPSLA Proceedings', pp. 411–428.
- Hartmann, S. & Roddick, J. (2004), 'The first asia-pacific conference on conceptual modelling (call for papers)', <http://apccm.massey.ac.nz/apccm04/>.
- Hovemeyer, D. & Pugh, W. (2004), Finding bugs is easy, in 'OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications', ACM, pp. 132–136.
- Jackson, M. (2001), *Problem Frames: Analyzing and Structuring Software Development Problems*, Addison-Wesley.
- Jencks, C. (1987), *The Language of Post-Modern Architecture*, Academy Editions.
- Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M. & Irwin, J. (1997), Aspect oriented programming, in 'ECOOP Proceedings'.
- Leuf, B. & Cunningham, W. (2001), *The Wiki Way*, Addison-Wesley Publication Co. <http://wiki.org/>
- Lyotard, J.-F. (1979), *La Condition postmoderne: Rapport sur le savoir*, Collection: "Critique.", Minuit, Paris.
- Lyotard, J.-F. (1984), *The Postmodern Condition: A Report on Knowledge*, Vol. 10 of *Theory and History of Literature*, University of Minnesota Press. Translated from the French by Geoff Bennington and Brian Massumi.
- Lyotard, J.-F. (1992), From the postmodern condition, in A. Easthope & K. McGowan, eds, 'A Critical And Cultural Reader', Allen & Unwin.
- Martin, A., Biddle, R. & Noble, J. (2004a), When XP met outsourcing, in J. Eckstein & H. Baumeister, eds, 'Proceedings of the Fifth International Conference on eXtreme Programming and Agile Processes in Software Engineering'.
- Martin, A., Biddle, R. & Noble, J. (2004b), The XP customer role in practice: Three case studies, in 'Proceedings of the Second Agile Development Conference'.
- Moore, I. & Pryce, N. (2005), 'Scrapheap challenge: A workshop in post-modern programming', Workshop at OOPSLA 2005, San Diego. <http://postmodernprogramming.org/>.
- Noble, J. & Biddle, R. (2002), Notes on postmodern programming, in R. Gabriel, ed., 'Proceedings of the Onward Track at Oopsla 02, the ACM conference on Object-Oriented Programming, Systems, Languages and Applications', <http://www.dreamsongs.org/>, Seattle, USA, pp. 49–71.
- Noble, J. & Biddle, R. (2004), Notes on notes on postmodern programming: radio edit., in 'OOPSLA Companion, proceedings of the Onward! stream', pp. 112–115.
- Opdyke, W. F. & Johnson, R. J. (1990), Refactoring: An Aid in Designing Application Frameworks, in 'Symposium on Object-Oriented Programming Emphasizing Practical Applications', ACM-SIGPLAN, pp. 145–160.
- Parnas, D. L. & Clements, P. C. (1986), 'A rational design process: How and why to fake it', *IEEE Transactions on Software Engineering* **12**(2), 251–257.
- Pearce, D. & Noble, J. (2006), Relationship aspects, in 'Aspect-Oriented Software Development (AOSD)'.
- Poppendieck, M. & Poppendieck, T. (2003), *Lean Software Development: An Agile Toolkit for Software Development Managers*, Addison-Wesley.
- Portland Patterns Repository (2005), 'You Aren't Extreme — Portland Patterns Repository', <http://c2.com/cgi/wiki?YouArentExtreme>. [Accessed November 2005].
- Raymond, E. S. (2001), *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, O'Reilly & Associates.
- Raymond, E. & Steele, G. L. (1993), *The New Hacker's Dictionary*, second edn, MIT Press.
- Rinard, M., Cadar, C. & Nguyen, H. H. (2005a), 'Exploring the acceptability envelope', Presentation to the OOPSLA 2005 Onward! Stream.
- Rinard, M., Cadar, C. & Nguyen, H. H. (2005b), Exploring the acceptability envelope, in 'OOPSLA Companion, proceedings of the Onward! stream'.
- Robinson, H., Hall, P., Hovenden, F. & Rachel, J. (1998), 'Postmodern software development', *The Computer Journal* **31**, 363–375.
- Sanrio Company, Ltd. & Sanrio, Inc (2005), 'The official Sanrio website. home of Hello Kitty', <http://www.sanrio.com/>.
- Schwaber, K. & Beedle, M. (2001), *Agile Software Development with SCRUM*, Prentice-Hall.
- Shalit, A. (1992), *Dylan: an object oriented dynamic language*, first edn, Apple Computer, Inc.
- Shalit, A., Moon, D. & Starbuck, O. (1996), *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*, first edn, Addison-Wesley.
- Stallman, R. M. (1981), EMACS the extensible, customizable self-documenting display editor, in 'Proceedings of the ACM SIGPLAN SIGOA symposium on Text manipulation', pp. 147–156.
- Stamper, R. (2001), Organisational semiotics: Informatics without the computer?, in K. Liu, R. Clarke, P. B. Andersen & R. Stamper, eds, 'Information, organisation and technology: Studies in organisational semiotics', Kluwer Academic Publishers, pp. 115–171.
- The Agile Alliance (2001), 'The Agile manifesto', <http://agilemanifesto.org/>.
- The Joint Task Force on Computing Curricula (2001), 'Computing curriculum 2001', *Journal on Education Resources in Computing* **1**(3es), 1.

Vlissides, J., ed. (1998), *Pattern Hatching: Design Patterns Applied*, Addison-Wesley.

Waugh, E. (1943), *Scoop*, Penguin Books.

Wikipedia (2005), 'Postmodernism — wikipedia, the free encyclopedia', <http://en.wikipedia.org/>. [Accessed November 2005].

Wirfs-Brock, R., Wilkerson, B. & Wiener, L. (1990), *Designing Object-Oriented Software*, Prentice-Hall.

Yourdon, E. & Constantine, L. L. (1979), *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, facsimile edn, Prentice Hall.