Notes on Postmodern Programming

James Noble, Robert Biddle
Computer Science,
Victoria University of Wellington, New Zealand.
{robert,kjx}@mcs.vuw.ac.nz

March 23, 2002

0 Manifesto

The ultimate goal of all computer science is the program. The performance of programs was once the noblest function of computer science, and computer science was indispensable to great programs. Today, programming and computer science exist in complacent isolation, and can only be rescued by the conscious co-operation and collaboration of all programmers.

The universities were unable to produce this unity; and how indeed, should they have done so, since creativity cannot be taught? Designers, programmers and engineers must once again come to know and comprehend the composite character of a program, both as an entity and in terms of its various parts. Then their work will be filled with that true software spirit which, as "theory of computing", it has lost. Universities must return to programming. The worlds of the formal methods and algorithm analysis, consisting only of logic and mathematics, must become once again a world in which things are built. If the young person who rejoices in creative activity now begins his career as in the older days by learning to program, then the unproductive "scientist" will no longer be condemned to inadequate science, for their skills will be preserved for the programming in which they can achieve great things.

Designers, programmers, engineers, we must all return to programming! There is no essential difference between the computer scientist and the programmer. The computer scientist is an exalted programmer. By the grace of Heaven and in rare moments of inspiration which transcend the will, computer science may unconsciously blossom from the labour of the hand, but a base in programming is essential to every computer scientist. It is there that the original source of creativity lies.

Let us therefore create a new guild of programmers without the class-distinctions that raise an arrogant barrier between programmers and computer scientists! Let us desire, conceive, and create the new program of the future together. It will combine design, user-interfaces, and programming in a single form, and will one day rise towards the heavens from the hands of a million workers as the crystalline symbol of a new and coming faith.

1 To Our Reader

These notes have the status of "Letters written to ourselves": we wrote them down because, without doing so, we found ourselves making up new arguments over and over again. When reading what we had written, we were always too satisfied.

For one thing, we felt they suffered from a marked silence as to what postmoderism *actually is* [75, 9]. Yet, we will not try to define postmodernism, first because a complete description of postmodernism in general would be too large for the paper [60, 43, 62, 71], but secondly (and more importantly) because an understanding of postmodern programming is precisely what we are working towards.

Very few programmers tend to see their (sometimes rather general) difficulties as the core of the subject and as a result there is a widely held consensus as to what programming is really about. If these notes prove to be a source of recognition or to give you the appreciation that we have simply written down what you already know about the programmer's trade, some of our goals will have been reached.

2 On Our Ability To Do Much

We are faced with a basic problem of presentation. What we are really concerned about is the composition of large systems, the text of which may occupy, say, a significant fraction all the digital storage media in the known world [72].

Our basic problem is simply the success of modern computer science. History has shown that this truth is very hard to believe. Apparently we are trained to expect a "software crisis", and to ascribe to software failures all the ills of society: the collapse of the dot-com bubble [27, 30], the bankruptcy of Enron [49], and the millennial end of the world [76].

This corrosive scepticism about the achievements of programming is unfounded. Few doom-laden prophesies have come to pass: the world did not end with fireworks over the Sydney harbour bridge, and few modern disasters are due to software. To consider just two examples: the space shuttle crash was not caused by software — indeed, Feynman praises the shuttle software practices as exemplary engineering [23]; and the Dot-Com Boom (like the South Sea Bubble) was not caused by failure of technology, but the over-enthusiasm of global stock markets.

In short, one cannot be woken up in the morning, travel to work, listen to radio or music; watch television; play games; speak or TXT down the 'phone; read newspapers or books; write conference papers, journal articles, government or corporate reports; save or spend money; buy food, cook it, order or pay for it at restaurants ranging from McDonalds to the Ritz; without every activity critically depending upon the results of programming. These programs are not perfect: but neither are they the complete, expensive failures beloved of armchair critics, whose behaviour belies their own rhetoric whenever they fly across the Atlantic in automated aircraft to speak at conferences and then use Internet banking to check their accounts. The measure of software is our irritation at its failures, not our surprise that it works at all.

Summary: as quick-witted human beings we have built very large computer systems and we had better learn to live with them and respect their limitations while giving them due credit, rather than to try to ignore them, so that we will be rewarded by continued success.

3 On the Notion of Program

The object of study of computer science is the program — somehow cobbled together by old-time programmers; measured, metricated, analysed, theorised, critiqued, tested, compiled, optimised, and often ignored by modern computer science. We consider that the term "program" is both too big and too little for postmodern computer science.

"Program" is too big because often we are working on parts of programs: objects, functions, classes, components. We may have no idea which (if any) program these parts will end up part of.

"Program" is too small because often we are working on multiple programs: perhaps the small component is to be part of many large programs; perhaps there is a framework or library that will be reused many times; perhaps our program must communicate with other programs, who in their turn communicate with yet more programs, so the subject of our endeavour is somewhere this interconnected network.

Large or small, the quality to which we refer is perfectly precise. Like bad art, we know it when we see it. Still, it cannot be named.

A word which we most often use to talk about programs is "component".

Yet, a component can only be a subpart, not a whole.

Another word which we use to talk about programs is "system".

A system could be large, or small, but includes the strong connotation of systematic, systems theory, system thinking, that the system is organised, rationally subdivided, structured recursively into a tree of modules, modern. The word "system" is too enclosed.

The word "algorithm" is often claimed as the central concept of computer science [34]

"Algorithm", however, leaves out large amounts of the discipline of programming: components, patterns, protocols, languages, data structures [74].

The word "software" is reminiscent of undergarments.

The phrase "*the software without a name*" could capture precisely what we wish to address. Unfortunately, experience teaches us that this software would soon have a name: "*SWAN*".

Thus, we have retained the word program, but treat it as under erasure, as meaning whatever program, program subcomponent, or supersystem we happen to be working on at the time.

4 On Pervasive Heterogeneity

Modern computer science dreamed of the personal computer: one machine usable by one person running one application written in one language. The personal computer was "the computer you could unplug".

We have progressed far beyond the modern dream. There is a "Computing Rainbow" [32] — an interdependent system of global computation with a multitude of machines supporting many languages, applications, and users, with heterogeneous architectures and differing capacities, costs, prices, and ownership. You cannot unplug computers even if you could want to: why would you want to avoid reading email from your friends; disconnect the full authority digital engine controllers from the turbofans carrying the airliner; or remove the cochlea implants correcting your congenital deafness?

Each of these subareas of programming has their own concerns, forces, difficulties, problems. Computer science is itself fragmented, although some concerns cut across several areas. But this heterogeneity does not operate only at the abstract level of the field as a whole; rather programs themselves are increasingly heterogeneous. A program may include a PalmPilot client which interfaces directly to an 360-architecture mainframe; Windows XP microcomputers may feed information into a Sun minicomputer; one division of a company may run Compaq VMS systems while another runs IBM AS/400 or HP7000. Programs have to federate across diverse systems, without any common language, protocol, or necessarily even character set in common.

5 On Abstraction

At this stage we find it easy to say something about the role of abstraction, partly because it permeates the whole subject.

Modern computer science describes the relationships within programs as "abstractions" — we may say an object in a program is an abstraction of the real world [8].

Computer scientists and mathematicians are familiar with abstractions: for example, a stack is an abstraction that might be implemented by an array, a pointer, and some executable code; the stack is an abstraction because it elides many of the details of actual implementation [16].

Unfortunately, it doesn't seem to make much sense to say that a Bovine object in a program is an "abstraction" of a real cow in a farm in this way: it doesn't make sense to say that the object in the program is "implemented" by a cow in reality, or that the objects in the program are special kinds of cows which do not eat, excrete, or expire. Alternatively, following Plato, we could have an abstraction of a cow as the "ideal, immutable, eternal form" of a cow, perhaps corresponding to a Cow *class*, but, again, this kind of abstraction is not a good description of the relationship between the cow object and the real cow [66].

Postmodern computer science proposes a range of different descriptions of the relationship between programmatic object and external object. For example, that this kind of relationship can be seen as semiotic: that is, the object in the program can be seen as a *sign* of the object in the world [53, 4, 3, 2]. Unlike abstractions, which can be reasoned about using deduction (from causes to effects), signs are effectively implications, and are modelled using abduction (reasoning from effects to causes) [20].

Semiosis may also support developments of a theory of debugging (determining bugs from symptoms); of analysis (determining the programs from requirements); and a metatheory of design (determining patterns, algorithms, structures from concrete programs) [52].

6 On Requirements

Postmodern computer science holds that no requirements can be both complete and consistent: you have to pick one.

Descriptivists, postmodernists choose completeness over consistency. In analysing a system (for example with Usage-Centered Design [13]) we may consider multiple users providing multiple requirements, and distinguish between actors who will finally use the system, customers who pay for it, clients who commission it, and stakeholders who wish they were involved — all of whom may be in conflict. Postmodern analysis uses techniques to handle inconsistency, such as iteration on designs, prioritising, and lying where necessary [59].

In contrast, modernists choose consistency over completeness. To perform any traditional formal analysis (without resorting immediately to modal logics) a description has to be consistent: before forming such a description, much information must be elided to ensure this consistency — adopting such a frame of reference necessarily excludes information that lies outside. Of course, a consistent definition has the great advantage of automated or manual checking, often the aim of the modernists — but as postmodernists we are willing to coopt their techniques whenever we feel they are useful.

Computer scientists with a formal bent often claim than design and implementation are an exercise — in the refinement calculus you can gradually transform a problem statement into a program, maintaining correctness at every step: problem frames have similar potential. From a postmodern perspective, however, such a definition is not a problem statement, but rather an abstract definition of a solution — where refinement simply makes a solution more concrete. Similarly, Jackson's problem frames are selected to match particular solutions, rather than problems per se: one fits problems to frames, rather than adjusting frames to enclose problems [42].

Corollary of this section: Formal analysis can be used to show the absence of bugs, but never to show the correctness of the specification. Or, to quote Alan Perlis: *"102. One can't proceed from the informal to the formal by formal means"* [57].

7 No Big Picture

A key characteristic of postmodernism is the absence of a "grand narrative" [62, 48, 60]. Where once the majority of the world would have believed in God or Marx, where architecture was simply building steel framed rectangular glass boxes, where music was constrained to the twelve-tone row, where citizens of a country all spoke the same language and supported the same cricket teams, we may now contemplate with fear a sea of chaos, perhaps held together by unenlightened self-interest.

Consider the Internet as we know it: a connection of loosely coupled computer systems, of varying capacities, architectures, ownerships, costs, and sizes. It connects every conceivable variation of every operating system and every computer. It speaks many network protocols (HTTP of various versions and brands; Telnet; POP; IMAP; NETBUI; AFS; SMB; ...); and through hardware software gateways, the systems and protocols of what appears to be the Internet are in fact unlimited. It can even reach computers that have been obsolete and no longer exist: retrocomputing lives through emulation and lives on the Internet.

Even the user experience of the World Wide Web is extremely diverse: every web site has its own design, its own interaction style, its own personality, with no commonality other than the menu bar provided by an individual's browser, one of many available, and customisable on a whim.

Compare this all with representations of a computational "infosphere" in popular science fiction — such as *The Matrix* [69] or *Neuromancer* [33]. These are typically modern in character, working in a complex but coherent way, and presenting a uniform interface: the "Matrix" presents a reslistic single graphical presentation common to all users. Ironically, the postmodern Internet is more real than these fantasies; and there is no one viewpoint on the Internet, and there may be no commonality between two web sites even if hosted on the same server and designed by the same people.

Although it clearly developed from the original success of the modern design of the Arpanet, the success of the Internet now is postmodern in character. But it is success. And the tolerance of eclectic diversity is a key cause of the success of the internet: it has allowed growth and interaction which instead of isolation and alientation.

For postmodern programming, the absence of an overarching grand narrative means eclectic tolerance in programming terms:

- There is equal acceptance of high and low culture: Visual Basic and Haskell are equally of interest, as there is no reason to applaud the one and disparage the other.
- The past is just another part of the present — programs can call on elements of modernism, either aesthetics or technology, and combine them together in equal measure. As ancient computers live through emulation on the Internet, so ancient programs and languages can live in connection with programs not yet written.
- Programs can exhibit "faults in construction" that would be forbidden by a modernist approach.

- Programming techniques (such as design patterns) and systems (such as Aspect/J [44], or even the continuation code sections in literate programming [45]) explitly support program organisation involving communicating diverse elements.

Without a grand narrative, there will not be one common way to program, or even one common kind of interface between programs. The alternative is the postmodern multidimensional organisation encompassing many little narratives.

In practice, narratives may grow and shrink, reflecting the exercise of power (especially by monopolist organisations) and the development of communities (especially where cooperation is mutually beneficial).

Moreover, there are many kinds of narrative in programming, and systems may have a postmodern character in some aspect but modern character in others. For example the Microsoft Common Language Runtime [35] is postmodern in that it supports a large number of programming languages — modern (C), postmodern (Perl), and historical (COBOL), high culture (Haskell) and low (Visual Basic), with access to low-level features as necessary (a modernist would consider this a fault). However, it is modern in that it achieves this by enforcing a common bytecode format — indeed, a particular subset of the format, and that it deploys the apparatus of power (verifiers, compliance kits, bytecode type checkers, developer certification, code component certificates) to enforce the commonality.

8 On Modular Components

Modularity, and interchangeable modular components are a key component of the modernist approach in software, as in architecture, marketing, production, and elsewhere.

Postmodernism admits modernism as one mode of expression, so modular or generated components (as with other modern techniques or tools) can readily be used to postmodern ends.

Consider for example the Sydney Opera House. This building is now symbolic of Sydney and Australia, but was originally sketched on the back of a napkin by Jørn Utzorn. The key feature of the Opera house are the shell-like roofs above the Opera Theatre and Concert Hall (a design which is postmodern because it has nothing to do with the function of the concert halls below).

The construction of the Opera House is only possible because the distinctive shells come from a single geometric spherical section. This was not part of the napkin design — yet according to Utzorn, this "solves all the problems of practical construction by opening up mass production" both for the tiles on the roof, and the ribs supporting them. Without mass production of modular components the Opera House could not have been completed.

9 No Metaphor

Postmodern programming rejects overarching grand narratives.

As a result, it favours descriptive reasoning rather than prescriptive. Rather than working top down from a theory towards practice, postmodern programming theories are built up, following practice. Moreover, theory follows practice on a case-by-case basis — "the world is all that is the case" [75].

Note that here we don't necessarily mean "theory" in the mathematical sense of theoretical computer science. Rather, we mean theory in the traditional speculative sense that serves to help us organise our experiences. For example, theories of how best to program would include stepwise refinement, object-orientation, and pattern languages.

Postmodern programming limits the scope of theory (and formalism) to particular "little narratives" — conditions where that theory is applicable, or is generated by the practice. This limits the kind of questions that will be asked of theory, and theory's position within the discipline as a whole.

Many metaphors have been adopted to describe programming: computer *science* (hypotheses, experiments, research); software *architecture* (plans, building, implementing); software *engineering* (design, verify, construct); and we may see programs as *literature* (write programs, literate programming); programs as *evolutionary biology* (program evolution, cellular automata); programs as *neurobiology* (artificial neural networks); programs as *mathematics* (programs as theorems, as proofs, as type systems).

Within modern computer science (following modern architecture, or disciplinary inadequacy due to the low status of computer science departments in many universities) there is an intellectual posture that accepts metaphors from other disciplines uncritically [63], without providing arguments as to why that metaphor should be applicable [77]. In general, this is the result of the modern grand narrative: computer science must conform to some theory, where that theory is carried by metaphor: the program as proof, as bridge, as house, as city.

Postmodern computer science tends to eschew metaphor — rather, in place of a metaphor we have a past. This is true on both the large scale (the discipline as a whole) and the small scale (individual programmers and programs). Postmodernism is often descriptive: recording the state of the world, rather than presenting some grand theory. Writing programs follows reading programs, because postmodern programming is extension, recovery, reuse, rather than creating masterpieces from nothing. Theory follows practice, because we aim to understand the world as it is, rather than remake it from scratch with a genesis device [64].

Our view is that computer science has "come of age". Computer Science is sufficient for itself: albeit as an 'unrestricted science' from where investigators must be prepared to follow their problems into any other science whatsoever (Pantin, quoted by Becher & Trowler,[5, p.32]). That is, we think it sensible that related disciplines are applied to their domains, so physics is used to address the design of semiconductors, statistics to analyse web server performance, accounting to study e-commerce, semiotics and psychology to drive human interface design, or linguistics to categorise Visual Basic programs. Of course we should be prepared to learn from many other disciplines. But the program itself the ultimately the subject of computer science itself.

10 No Future

What is **post-**modernism? And where does it lead? How can something be after what is **modern**?. Isn't modern what we have *today?*

Modernism is a term used to describe a range of developments in architecture, literature, philosophy, and then society generally. Postmodernism is what comes after modernism. The question is, does postmodernism:

- **replace** modernism?, or
- **fulfil** modernism?

Inasmuch as there is an answer, it is both. (This is the standard postmodern answer to any question. 'Tsall good). Postmodernism is a replacement for modernism because the postmodern theories or practices replace the modern. Postmodern architecture has replaced modern architecture; postmodern fiction has replaced modern fiction; postmodern programming languages (Perl, late C++) replace modern programming languages (Pasal, ANSI C).

But postmodernism (or postmodernity, the society and culture that follows after modernity) is simultaneously the fulfilment of modernism. Without the technology developed by modernity, there could be no postmodernity or postmodernism. Thus, Extreme Programming, for example, aims to replace modern and late-modern methodologies (e.g. Responsibility Driven Design or the "Booch" methodology, and the Rational Unified Process or the OPEN process (now deceased)) [6, 73, 7, 47, 36]. On the other hand, XP also claims to be the fulfilment of a number of modern movements: including rigorous testing, consistent coding and naming style, and late-modern programming languages and environments (e.g. Smalltalk) perhaps with postmodern extensions (JUnit, the Refactoring Browser). Similarly, postmodern programming does not reject but rather embraces elements that are themselves the ultimate products of modern development.

11 Perl, The First Postmodern Programming Language

What is a postmodern programming language? Or, what is a modern programming language? The second question is easier to answer than the first: a modern programming language supports a single (modern) style of programming, based on recursive decomposition of both code and data.

Modern programming languages can themselves vary in a number of ways. Common Lisp, APL, and Smalltalk, for example, are based on difference approaches to programming, but all are modern, and all rely on extensive support for their programming theories. Pascal, Oberon and Scheme and Self are more minimalist counterparts, but also modern.

Larry Wall explains how his design of Perl was explicitly based on a postmodern approach. This explanation shows how postmodernism, easily misuderstood or disregarded by pragmatists, is responsible for a programming language highly prized and defended by those same pragmatists. Wall's reasoning deserves to be read in in its entirety, but the key point is that the design was not based on any grand narrative, but on a case-by-case basis: [70].

> *When I started writing Perl, I'd actually been steeped in enough postmodernism to know that that's what I wanted to do. Or rather, that I wanted to do something that would turn out to be postmodern, because you can't actually do something postmodern, you can only really do something cool that turns out to be postmodern. Hmm. Do I really believe that? I dunno. Maybe. Sometimes. You may actually find this difficult to believe, but I didn't actually set out to write a postmodern talk. I was just going to talk about how Perl is postmodern. But it just kind of happened. So you get to see all the ductwork.*
>
> *Anyway, back to Perl. When I started designing Perl, I explicitly set out to deconstruct all the computer languages I knew and recombine or reconstruct them in a different way, because there were many things I liked about other languages, and many things I disliked. I lovingly reused features from many languages. (I suppose a Modernist would say I stole the features, since Modernists are hung up about originality.) Whatever the verb you choose, I've done it over the course of the years from C, sh, csh, grep, sed, awk, Fortran, COBOL, PL/I, BASIC-PLUS, SNOBOL, Lisp, Ada, C++, and Python. To name a few. To the extent that Perl rules rather than sucks, it's because the various features of these languages ruled rather than sucked.*

Perl is a success, and there is no grand narrative for its design. However, over time Perl is getting more modern, accumulating all the trappings of a modern language: objects, packages, namespaces, syntax, etc. Again, this is an example of the post-modern trait of absorbing technology from modernism while putting it to a rather different use.

While Wall's rationale is explicit, other languages can place a good claim to be considered postmodern, at least in some aspects. Hypercard and Visual Basic extend programming to include forms of multimedia and graphical user interface (again without getting some religion: Prograph is modern in its insistence on the primacy of visual syntax). Intercal must be considered as a post-modern language (mostly for non-technical reasons).

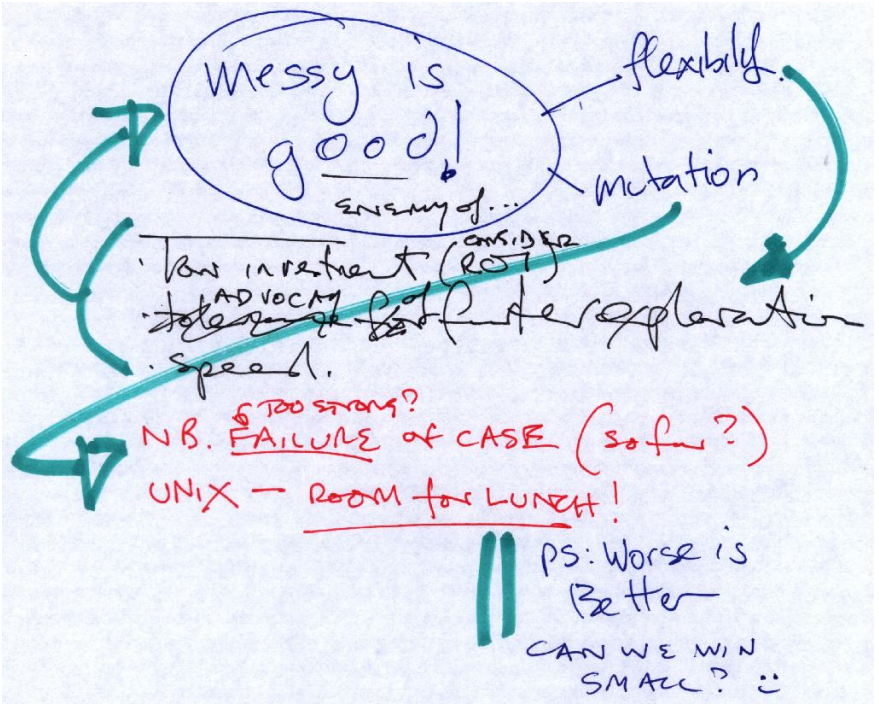
PL/I could be considered post-modern, as it was designed to support Fortran, Algol, and even COBOL programming styles, although (unlike Perl) it attempted to bring them all within a modern unified framework. This mixed result is involved in Dijkstra calling PL/I a fatal disease [17]:

> PL/I --"the fatal disease"-- belongs more to the problem set than to the solution set.
>
> It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration.
>
> The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence.
>
> APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums.

C++ is an interesting case. Early C++ — C with Classes, the C++ used in the *Design Patterns* book is essentially a modern object-oriented programming language: classes are the primarily (and only) modelling technique, no-one seriously advocates the procedural features of a language — or rather, these "faults in construction" are tolerated as faults within the modern narrative. As C++ evolves, into what we call "Late C++" (templates, exceptions, dynamic casting) the discourses surrounding C++ change also — leaving the object-oriented grand narrative, and becoming postmodern, building a theory of multi-paradigm design and programming upwards from the language features [14, 65, 46]. While *Design Patterns* (as one data point of C++ circa 1994) has essentially no examples of C++ free functions, by the late 2000 Late C++ is clearly advocated as a postmodern language.

Finally, it is interesting to consider Java vs. C♯. Both are arguably postmodern languages, although less so than Perl, and with stronger streaks of modernism, especially in the one-language rhetoric surrounding Java, matched by the CLR rhetoric surrounding C♯. There are no significant technical differences between the two languages — both with C++ syntax, somewhat moderated by the Pascal tradition, with a ersatz-Smalltalk object model and a handful of Modula-3 thrown in for concurrency and modularity. The key reason these languages are postmodern is that they cannot be considered against technical criteria: comparing them is like comparing Pepsi and Coke: you don't drink the cola — you drink the advertising [67].

12 Messy Is Good



13 A First Example of Scrap-Heap System Construction

In the section "No Metaphor" we will have stressed that postmodernism has a past, and that this past is reflected in both the structure of the discipline and the practice of programming. In particular, this past exists as a large number of existing programs that the postmodern programmer can scavenge through and reuse.

Instead of presenting (as a ready-made product) what we would call a scrap-heap program, we are going to describe in detail the process of creating such a program. We do this because many programs are just there: they do not have to be made, and the kind of programs we are particularly interested in are those which we feel to be comfortably outside our powers of construction and conception.

The task is to instruct a computer to print a table of the first thousand prime numbers, 2 being considered the first prime number.

To write this program, we first connected our computer to the Internet, downloaded some music from Napster, and then read our email. (You have to receive email to perform a workday [11]). We received 25 pieces of email of which 16 were advertisements for Internet pornography, administriva, or invitations to invest in Nigerian currency trades. After dealing with this email, we typed "calculate prime numbers" into Google. This found several web sites regarding prime numbers, and some more pornography. After a while, we were interrupted, and so moved on to the prime number web sites. In particular, `http://www.2357.a-tu.net` includes a the "ALGOMATH" C library for calculating prime numbers; another site included an EXCEL macro which was top complex to understand. Although we had not programmed in C for years, after downloading and compiling the library (by typing `make`), we noticed the documentation included the following program:

```
int *pointer , c=0;

if((pointer = am_primes_array(4, 3)) == NULL)
    printf("not enough memory\n");

while( *(pointer+c)){
                    printf("%d\n",*(pointer+c));
       c++;
   }

return;
```

We cut and pasted this program into a file and compiled it several times, having to add a few extra lines (e.g. `main () {`). Eventually we ran it, and indeed it appeared to generate three prime numbers larger than four. We edited the parameters to `am_primes_array` to `(2,1000)`, and then ran the output through `wc -l` to check that it had printed 1000 numbers.

Here we have completed what we announced at the beginning of this section, viz. "to describe in very great detail the composition process of such a [postmodern] program".

14 We're all Devo

- The Open toolbox of techniques catalogues development practices without giving any information about how they would fit into the development lifecycle [40].
- Foote's Big Ball of Mud pattern language can be read as advocacy, apologia, or critique of unorganised development, a defence of postmodern programming, or a parody of patterns in general [25]
- Perl [70]
- Usage-Centered Design deconstructs the subjects of our programs into many individual user rôles, and the intentions (desires) of those subjects into very many essential use cases [13].
- Groves has demonstrated how the refinement calculus can describe maintenance, modification, and refactoring, opening the way for *Extreme Formalism* [38].
- Scripting Languages (e.g. Tcl, Ruby, JavaScript) are designed to be only some part of the program, and are never written in themselves [54].
- *Lisp: Good News, Bad News, How to Win Big* offers a last-ditch defence of late modernism: *"I think there will be a next Lisp. This Lisp must be carefully designed, using the principles for success we saw in worse-is-better."* [28].
- Dream Machines and Computer Lib present the Digital Equipment Corp. line of computers in high postmodern style (a large-format comic-book) [51].
- Jackson's *Software Requirements & Specifications* provides a (postmodern, post-structured) dictionary of development terminology, while *Problem Frames* recontextualises his earlier Structured Design and Structured Programming Methods as one technique amongst many [41, 42].
- The Microsoft-brand Common Language Runtime supports many different programming languages with a common (late-modern) execution framework [35].
- The *Rational Software Process — How and Why to Fake it* describes why unifying irrational processes is more rational than Unified Processes [56, 47].
- Aspect- and Subject- Oriented Programming in Aspect/J, Hyper/J, and Composition Filters breaks down the recursive structure of object-oriented programs to introduce a multidimensional subdivision [44, 39, 1]. The Aspect Browser visualises the topics of discourse within program texts [37].
- Literate programming's continuation code sections deconstruct the rigorous recursive structure of Pascal programs so that parts of procedures can be presented in a order that suits explanation and documentation [45].
- Intentional Programming separates syntax and semantics, deconstructing languages so programs can be written in any language or style [15].
- Evolutionary computation, neural networks, and cellular automata reject large scale descriptions in favour of local action.
- *A Small Matter of Programming* [50].
- Anything to do with Spreadsheets [55, 61]
- Agile Methodologies and Extreme Programming. Development proceeds incrementally, customised to suit the occupational culture [12]. Note that Extreme Programming still insists on a grand narrative, in contrast to Agile development generally [26].
- Open Source and Mob Software development replace centralised development by a single company with mongolian hordes of programmers giving their time free across the internet [31, 58].
- Mr Bunny's Big Cup O' Java — Farmer Jake gets to places Niklaus Wirth can only dream about [22].
- Minimal Manuals — practical, descriptive information written on cards, with no precise ordering or overarching theory of operation [10].

15 Small Stories of Devotion

> . . . We have the whole world
> in our hands, smiles one. What the hell
> are we going to do with it, laughs the other.
>
> *Small Stories of Devotion*
> Dinah Hawken

Gamma, Helm, Johnson and Vlissides *Design Patterns* remains for us, at least, a crucial text. It's not possible to give a single date for the start of postmodernism in computer science — as with other postmodernisms, "it seems to have slunk over the horizon" [62, p.158]. If pressed, we would choose OOPSLA 1994, where the *Design Patterns* book first became openly available. In a lovely postmodern irony, the book is copyrighted 1995.

Reflecting upon *Design Patterns* from the distance of the best part of a decade (an octade?) we should pause, once again to be surprised at its continued successes, not irritated over its failures. *Design Patterns* has sparked a number of imitators: many mediocre, some less so, none as successful as the original, along with several edited collections and a multinational multiannual conference metaseries. The idea of an object-oriented design pattern, of the kind described in the book, is now accepted throughout computer science practice, incorporated into the libraries and documentation for emerging programming languages such as Java and C♯, and taught routinely in most undergraduate programming curricula. The breath of the authors' vision is clear that in the last eight years, although many patterns have been written on a variety of topics, less than ten additional object-oriented design patterns have been found that are of a piece with the original twenty-three.

There have been a number of more-or-less organised critiques of *Design Patterns*, arguing that the patterns approach betrayed the future of modern computer science (a conclusion with which we agree). The nature of this betrayal varies, of course: some arguing that patterns remove or resist formalisation, taking us to hell in a phenomenological handbasket [68, 21]; others that design patterns have corrupted the Alexanderian heritage of a pattern language into which all the patterns must fit [24, 29].

We contend that *Design Patterns* is postmodern precisely because it does not fit into an overarching prescriptive narrative of design: programmers are free to use or not use patterns as they see fit, as one of many techniques at their disposal. This makes it easy to adopt design patterns whatever personal or corporate philosophy you espouse. Precisely because patterns are small independent narratives, supported by arguments made on a case-by-case basis in favour of certain designs, it is easy to learn patterns piecemeal. The structure of the book is essentially arbitrary, although there are a number of distinct and subtle relationships between individual patterns [52]. *Design Patterns* certainly builds on modern techniques (cohesion and coupling, modern languages such as C++ or Smalltalk; OMT design notation): but this is not problematic — modern technology often ends up in the service of postmodern aesthetics.

Finally, we consider *Design Patterns* to be postmodern because it is concerned with the practice of programmers working out their own designs, embodied within the programs that they create. The focus of the book is the artifacts themselves: programs, designs, code, treated as objects of intellectual

study and critique. But suffused all through the text, amid the concerns for pedagogy, efficiency, flexibility, and convincing argument, is the authors' clear respect for the topic of their discourses: their love of programs and programming.

> I think of the postmodern attitude as that of a man who loves a very cultivated woman and knows that he cannot say to her, 'I love you madly', because he knows that she knows (and that she knows that he knows) that these words have already been written by Barbara Cartland. Still, there is a solution. He can say, 'As Barbara Cartland would put it, I love you madly.' At this point, having avoided false innocence, having said clearly that it is no longer possible to speak innocently, he will nevertheless have said what he wanted to say to the woman: that he loves her, but he loves her in an age of lost innocence. If the woman goes along with this, she will have received a declaration of love all the same.
>
> *Reflections on the Name of the Rose* [19]
> Umberto Eco, 1985

References

[1] M. Aksit and A. Tripathi. Data abstraction mechanisms in SINA/ST. In *OOPSLA Proceedings*, 1988.

[2] Peter Bøgh Andersen. Computer semiotics. *Scandinavian Journal of Information systems*, 4:3–30, 1992.

[3] Peter Bøgh Andersen. *A Theory of Computer Semiotics*. Cambridge University Press, second edition, 1997.

[4] Peter Bøgh Andersen and Palle Nowack. Tangible objects — connecting informational and physical space. In Lars Qvortrup et al., editor, *Virtual Space: The Spatiality of Virtual Inhabitated 3D Worlds*. Springer-Verlag, 2001.

[5] Tony Becher and Paul Trowler. *Academic Tribes and Territories: Intellectual Enquiry and the Cultures of Discipline*. Open University Press, 2nd edition, 2001.

[6] Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 1999.

[7] Grady Booch. *Software Engineering with Ada*. Benjamin Cummings, 1983.

[8] Grady Booch. *Object Oriented Analysis and Design with Applications*. Benjamin Cummings, second edition, 1994.

[9] John Cage. *Silence*. Wesleyan Univ Press, 1973.

[10] John Carroll. *The Nurnberg Funnel: Designing Minimalist Instruction for Practical Computer Skill*. MIT Press, 1990.

[11] Patrick Chan and Carleton Egremont III. *Mr Bunny's Internet Startup game*. Addison-Wesley, 2000.

[12] Larry L. Constantine. *The Peopleware Papers*. Prentice-Hall, 2001.

[13] Larry L. Constantine and Lucy A. D. Lockwood. *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*. Addison-Wesley, 1998.

[14] James O. Coplien. *Multi-Paradigm Design for C++*. Addison-Wesley, 1999.

[15] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[16] Edsger W. Dijkstra. Notes on structured programming. In Ole-Johan Dahl, Edsger W. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming*. Academic Press, 1972.

[17] Edsger W. Dijkstra. How do we tell truths that might hurt? published as [18], June 1975.

[18] Edsger W. Dijkstra. How do we tell truths that might hurt? In *Selected Writings on Computing: A Personal Perspective*, pages 129–131. Springer-Verlag, 1982.

[19] Umberto Eco. *Reflections on the Name of the Rose*. Secker & Warburg, 1985.

[20] Umberto Eco. *Semiotics and the Philosophy of Language*. Indiana University Press, 1986.

[21] A. H. Eden, A. Yehudai, and G. Gil. Precise specification and automatic application of design patterns. In *1997 International Conference on Automated Software Engineering (ASE'97)*, 1997.

[22] Carlton Egremont III. *Mr. Bunny's Big Cup o' Java*. Addison-Wesley, 1999.

[23] Richard Phillips Feynman. *What Do You Care What Other People Think?: Further Adventures of a Curious Character*. W.W. Norton, 1988.

[24] Brian Foote. The show trial of the gang of four for crimes against computer science. Panel at OOPSLA 1999. `http://www.laputan.org/patterns/gang-of-four.html`, November 1999.

[25] Brian Foote and Joe Yoder. Big ball of mud. In Neil Harrison, Brian Foote, and Hans Rohnert, editors, *Pattern Languages of Program Design*, volume 4, chapter 29, pages 653–692. Addison-Wesley, 2000.

[26] Martin Fowler. The new methodology. `http://www.martinfowler.com/articles/newMethodology.html`, November 2001.

[27] David Futrelle. Enron contra. *CNN MONEY*, January 2002. Jan 25th. `http://money.cnn.com/2002/01/25/techinvestor/futrelle`.

[28] Richard P. Gabriel. LISP: Good news, bad news, how to win big. *AI Expert*, 6(6):30–39, 1991.

[29] Richard P. Gabriel. Back to the future: Worse (still) is better! `http://dreamsongs.com/NewFiles/ProWorseIsBetterPosition.pdf`, December 2000.

[30] Richard P. Gabriel. Wither software. `http://www.dreamsongs.com/NewFiles/WhitherSoftware.pdf`, March 2002.

[31] Richard P. Gabriel and Ron Goldman. *Mob Software: The Erotic Life of Code*. Dreamsongs Press, 2000.

[32] Richard P. Gabriel and Dave Thomas. Computing rainbow. Report of Feyerabend Workshop, `http://www.dreamsongs.com/Feyerabend/FeyerabendW4.html`, May 2001.

[33] William Gibson. *Neuromancer*. Ace Books, 1984.

[34] Les Goldschlager and Andrew Lister. *Computer Science : A Modern Introduction*. Prentice-Hall, 1982.

[35] John Gough. *Compiling for the .NET Common Language Runtime*. PTR PH, 2002.

[36] Ian Graham, Brian Henderson-Sellers, and Houman Younessi. *The OPEN Process Specification*. Addison-Wesley, 1997.

[37] William G. Griswold. The aspect browser. `http://www.cs.ucsd.edu/users/wgg/Software/AB/`, March 2002.

[38] Lindsay Groves. *Evolutionary Software Development in the Refinement Calculus*. PhD thesis, Victoria University of Wellington, 2000.

[39] Willian Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In *OOPSLA Proceedings*, pages 411–428, 1993.

[40] Brian Henderson-Sellers, Anthony Simons, and Houman Younessi. *The OPEN Toolbox of Techniques*. Addison-Wesley, 1998.

[41] Michael Jackson. *Software Requirements & Specifications*. ACM, 1995.

[42] Michael Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems*. Addison-Wesley, 2001.

[43] Charles Jencks. *The Language of Post-Modern Architecture*. Academy Editions, 1987.

[44] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP Proceedings*, pages 327–353, 2001.

[45] Donald Ervin Knuth. *Literate Programming*. CSLI Publications, 1992.

[46] Andrew Koenig. Idiomatic design. In *Addendum to the proceedings of the 10th annual conference on Object-oriented programming systems, languages, and applications*, pages 14–19. ACM Press, 1995.

[47] Philipe Krutchen. *The Rational Unified Process*. Addison-Wesley, 1999.

[48] Jean-François Lyotard. From the postmodern condition. In Anthony Easthope and Kate McGowan, editors, *A Critical And Cultural Reader*. Allen & Unwin, 1992.

[49] Julie Mason. Auditing software raised 'red alert'. *Houston Chronicle*, January 2002. Jan 25th. `http://www.chron.com/cs/CDA/printstory.hts/business/-1227548`.

[50] Bonnie A. Nardi. *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press, 1993.

[51] Theodor H. Nelson. *Computer Lib/Dream Machines*. Aperture, 1974.

[52] James Noble and Robert Biddle. Patterns as signs. In *ECOOP Proceedings*, 2002.

[53] James Noble, Robert Biddle, and Ewan Tempero. Metaphor and metonymy in object-oriented design patterns. In *Proceedings of Australian Computer Science Conference (ACSC)*. Australian Computer Society, 2002.

[54] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[55] Raymond D. Panko. What we know about spreadsheet errors. *Journal of End User Computing*, 10(2):15–21, Spring 1998.

[56] David L. Parnas and Paul C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12(2):251–257, 1986.

[57] Alan Perlis. Epigrams on programming. *ACM SIGPLAN Notices*, 17(9), September 1982.

[58] Eric S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly & Associates, 2001.

[59] H. Robinson, P. Hall, F. Hovenden, and J. Rachel. Postmodern software development. *The Computer Journal*, 31:363–375, 1998.

[60] Margaret Rose. *The Post-Modern and the Post-Industrial : A Critical Analysis*. Cambridge University Press, 1991.

[61] K. J. Rothermel, C. R. Cook, M. M. Burnett, J. Schonfeld, T. R. G. Green, and G. Rothermel. Wysiwyt testing in the spreadsheet paradigm: An empirical evaluation. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 230–239, June 2000.

[62] Stuart Sim, editor. *The Routledge Companion to Postmodernism*. Routledge, 2001.

[63] Alan Sokal and Jean Bricmont. *Intellectual Impostures*. Profile Books, July 1998.

[64] Jack B. Sowards. Star trek II: the Wrath of Kahn. Motion Picture.

[65] Bjarne Stroustrup. Why C++ is not just an object-oriented programming language. In *Addendum to the proceedings of the 10th annual conference on Object-oriented programming systems, languages, and applications*, pages 1–13. ACM Press, 1995.

[66] Antero Taivalsaari. Classes vs. prototypes: Some philosophical and historical observations. In James Noble, Antero Taivalsaari, and Ivan Moore, editors, *Prototype-Based Programming: Concepts, Languages and Applications*, chapter 1. Springer-Verlag, 1999.

[67] James B. Twitchell. *twenty ADS that shook the WORLD*. Three Rivers Press, 2000.

[68] Peter van Emde Boas. Resistance is futile; formal linguistic observations on design patterns. Technical report, University of Amsterdam, 1997.

[69] Andy Wachowski and Larry Wachowski. The Matrix. Technicolor 35mm Motion Picture, 1999.

[70] Larry Wall. Perl, the first postmodern computer language. `http://www.wall.org/ larry/pm.html`, Spring 1999.

[71] Nigel Wheale, editor. *The Postmodern Arts : An Introductory Reader*. Routledge, 1995.

[72] Winfried Wilcke. Computer architecture challenges in the next ten years. Keynote talk to Australian Computet Science Week 2002.

[73] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Prentice-Hall, 1990.

[74] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976.

[75] Ludwig Wittgenstein. *Tractatus Logico-Philosophicus*. Routledge and Kegan Paul, 1961.

[76] Edward Yourdon and Jennifer Yourdon. *Time Bomb 2000!: What the Year 2000 Computer Crisis Means to You!* Prentice Hall PTR, 1997.

[77] Liping Zhao and James O. Coplien. Symmetry in class and type hierarchy. In James Noble and John Potter, editors, *In Proc. Fortieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Conferences in Research and Practice in Information Technology. Australian Computer Society, 2002.