The Future of Programming

Peter Grogono

Department of Computer Science, Concordia University

1455 de Maisonneuve Blvd. West, Montréal, Québec H3G 1M8

E-mail: grogono@cs.concordia.ca

A Note to the Reader   An earlier version of this paper provided the basis for my talk *The Future of Programming* (Department of Computer Science, 6 November 1995). I am circulating this draft in response to requests for a written version of the talk. The paper is still in a very rough state and needs a great deal of work. Please do not make copies of it for your friends: tell them to ask me for copies. Also, please provide feedback: your comments are very welcome!

Abstract

Software development techniques are evolving slowly. It is widely held that technical improvements (new programming languages, specification techniques, and so on) will not have a dramatic effect on the reliability of software or the productivity of programmers. Proponents of this viewpoint look for changes in management practice to achieve significant improvements in productivity.

In this paper, we take a contrary view: technical improvements can, and probably will, change the nature of software development in radical ways. We survey applications of computers in areas other than software development and note a number of exciting advances. We then discuss ways in which these new approaches might be used by software engineers.

For other views of the future of software, see Lewis *et al.* (1995) and Leebaert (1995).

1   Introduction

It is common knowledge that computers lack common sense. We have all had the experience of receiving two subscriptions with different names or bills for zero amounts. Computer failures have become more serious now that they can give a lethal dose of radiation to a patient, or shoot down a civilian aircraft, or permit million-dollar thefts.

In the discipline of software engineering, software is called *robust*, or *resilient*, if it does not fail catastrophically in unanticipated circumstances. Although robustness is a much weaker requirement than common sense, it is a step in the right direction. It is not easy to design and build robust software, however, and much of today's software is notorious for its lack of robustness.

Software is "brittle" in the sense that even small variations in operating environment or user needs can cause software to fail completely. Such extremely tight tolerances make software

development a very exacting, very costly, and very time-consuming task. As society demands ever more complex systems, the difficulty and expense of developing adequate software becomes increasingly significant. We can improve the robustness of software by:

- using a fault-tolerant architecture where software monitors its own performance;
- allowing software to adapt or evolve as it monitors its operating environment and requirements; and
- rigorous testing to promote both fault-tolerance and adaptiveness.

There is promising evidence for each of these approaches in isolation. Our speculative goal is that a combination of several of these approaches will lead to resilient software and that such software will be easier and cheaper to develop. The purpose of our project is to investigate new approaches to the problem of creating robust software, to assess the feasibility of these approaches, and to develop the first prototypes of a new kind of software: software that is robust, resilient, and adaptive.

Research in a variety of disciplines, including astronomy, biology, chemistry, mathematics, medicine, and physics has exploited the capabilities of modern computers, often with spectacular results. In addition to contributing to these fields, this research has opened up new areas for investigation by increasing our understanding of complex systems. Ironically, one discipline that has not used the computer to make significant advances is computer science itself. Computer scientists have built tools for others to use, but have neglected their own tools.

The paper is organized as follows. Section 2 reviews the conventional approaches to software development, in which technical issues are seen as marginal and new styles of management are called for. Section 3 outlines a speculative analogy between software systems and physical systems. The analogy suggests some new approaches to software development. Subsequent sections of the paper do not depend on the arguments in this section. Section 4 contains most of the substance of the paper in the form of a catalog of software development techniques ranked roughly from conventional to radical. Section 5 is a tentative beginning; it outlines how some of the ideas of Section 4 might be applied to the construction of robust software. The goal of the "robust software project" is to provide substance and justification for this section. Finally, Section 6 offers some tentative conclusions.

2   The Conventional View

Studies have shown that that coding accounts for only 15% of a typical software development project. It is easy to infer that improving coding techniques can have at most a marginal effect — even if the coding time was reduced to zero, productivity would increase by a factor of $1/0.85$ or about 18%. This has led people to conclude, for example, that there is not much point in designing better programming languages.

The flaw in the argument is that it does not account for the influence of coding methods on the project as a whole. The process models that we use, such as the waterfall model and the spiral model, are based on standard programming languages and practices. Significant developments in implementation technology may lead to novel process models.

Expert software engineers have arrived at similar conclusions by different routes: they predict that technical improvements will yield only marginal improvements in software productivity and quality. Their predictions are based on the experiences of previous "breakthroughs" in the discipline: automatic, structured, object oriented, and visual programming; environments for software development; prototyping; automatic verification; artificial intelligence and expert systems; and others.

Brooks (1978), for example, argues that technical aspects, such as the choice of programming language, play a relatively small role in software productivity. Since the major problems in software development are managerial, significant improvements in productivity require better management techniques rather than better software tools. Brooks (1987) subsequently argued that software development is *essentially* hard, implying that there is "no silver bullet" — no way of substantially reducing the cost of software production. Technical innovations will affect only the *accidental* aspects of software development. (Brooks is using the words "essential" and "accidental" in Aristotle's sense. The *essence* of software building is the "mental crafting of the conceptual construct" (1978, page 209), but the implementation of the software is merely an *accident* or appurtenance.)

Parnas (1985) believes that software engineering is fundamentally different from software engineering. Engineering deals with artefacts such as loaded beams. As the load on the beam increases, the deflection of the beam increases according to a simple law until a certain load is reached, at which point the beam breaks. Software, however, does not deform continuously. Since programs typically contain many decision points, the output of a program is a complex and discontinuous function of its inputs. Reasoning about programs is therefore intrinsically more difficult than reasoning about structures and machines. Validation of software will always be hard, and there may even be limitations to the complexity of software that we can build.

The arguments of Brooks and Parnas obviously have some validity. Radical improvements in software productivity have not occurred in spite of promises from advocates of structured programming, object oriented techniques, and similar fads. Furthermore, software continues to fail in unexpected and unpredictable ways.

As a comparison, imagine a group of designers studying the stagecoach 120 years ago. After much discussion, they agree to concentrate all their efforts on improving the coach, since there is not much they can do to improve the horse. Analogously, designers of today show little interest in programming languages.

Our hypothesis is that current programming techniques are today's horses. They will soon be replaced by the software equivalent of the gasoline engine — new approaches to programming that allow robust software to be developed rapidly. Accordingly, we believe that there are solutions to the problems that Brooks, Parnas, and others believe to be inherent in software development. Furthermore, some of these solutions are not mere speculations, but have already proved themselves in areas other than software engineering. There is no shortage of innovative experiments: chaos theory (Gleick 1987), autonomous agents (Maes 1995), artificial life (Levy 1992; Kelly 1994), visualization (Brand 1987; Robertson, Card, and MacKinlay 1993), to mention but a few. As computer scientists, we have provided the means for others to explore these areas but, as software engineers, we have not attempted to exploit the new discoveries in our own work.

We believe that it is important to seek novel ways of creating software that is robust and adaptive. Whether there is a "software crisis" is debatable (Glass 1994), but there is certainly a need for software of higher quality. Software is already a major industry and is growing in importance. Increasingly, lives depend on robust software. Yet, as experts such as Brooks and Parnas have shown, traditional methods of software development do not yield robust products.

During 1984–94, the software industry expanded by almost 300%. In the USA, the software sector is larger than all but five manufacturing industries (Leebaert 1995). Canada is at the forefront of the software industry. Local companies such as Matrox, Metrowerks, SoftImage, and Visual Edge have sold their products to industrial giants such as Apple, IBM, and Microsoft. Canadian contributions are not always positive: errors in the Therac-25 control software caused several fatalities and received international attention.[1]

3   The Physical Systems Analogy

This section is perhaps the most speculative in the paper. It is also the least important: subsequent sections do not depend on it. For an extended discussion of the ideas outlined here, see Cohen and Stewart (1994).

During the last few decades, we have seen dramatic discoveries in the field of dynamic systems. In particular, the concept of "emergent behaviour" has slowly gained acceptance.

Conventional science is reductionist. Its goal is to explain each set of phenomena in terms of a lower-level, simpler set of phenomena. Biology is explained in terms of biochemistry, which is explained in terms of chemistry, which is explained in terms of physics, and so on. Ultimately, everything is explained in terms of fundamental particles — today, quantum mechanics.

There are two serious objections to the reductionist program. The first is that it is of little practical value. Although quantum mechanics is a supremely precise theory, it provides detailed, quantitative explanations of only the simplest phenomena. For anything more complex than a hydrogen atom, the calculations required by quantum mechanics are infeasible. It is futile to explain the behaviour of ants in terms of quantum mechanics.

The second objection to the reductionist program is that it is unnecessary. Recent discoveries show that simple situations may yield complex phenomena. A well-known example is the Miller-Urey (1953) experiment, which demonstrated that amino acids could be synthesized in quite simple environments, suggesting that the origin of life is not so improbable as previously believed. More recently, computer simulations have shown that evolution emerges in almost any environment that provides replication with variation and differential survival rates.

We make a tentative analogy between the failure of reductionism and the difficulties of software engineering. Our approach to software has traditionally been reductionist in nature. We explain systems in terms of their procedures, procedures in terms of their statements, statements in terms of machine instructions, and machine instructions in terms of gates. Of course, we have also learned the advantages of abstraction, in particular the important distinction between *what* a particular component does and *how* it performs the task.[2]

---

[1]Need a reference to Levenson.

[2]This distinction is probably over-rated. In practice, the step from "what" to "how" is infinitely divisible.

The software analogue of emergent behaviour does seem to exist. There are short programs that are hard to understand, such as the "gingerbreadman" (Peitgen and Saupe 1988, page 149). But there do not appear to have been any attempts to exploit emergent behaviour in software design. Emergent behaviour could have important consequences for software design because it would decouple the design of higher levels from the detailed operation of lower levels. A designer would require only that the lower levels satisfied general criteria, not that they provided a precise and detailed interface.

Applying our analogy, programmers are in the position of researchers in evolution who could make no progress without understanding all the details of the chemistry of RNA and DNA. In fact, evolution was well understood long before genetic engineering emerged as a discipline.

Figure 1 provides a summary of our analogy between physical systems and programming. The upper part of the table, above the double line, shows typical properties of the conservative approach. The lower part of the table gives typical properties of dissipative physical systems, and suggests properties of software that loosely correspond.

	Physical Systems	Software Systems
Conservative Systems	Linear	Exact
	Stable orbits	Loops
	Unstable	Brittle
	History sensitive	Static
Dissipative Systems	Non-linear	Fuzzy
	Limit cycles	Robust
	Self-stabilizing	Adaptive
	History insensitive	Evolving

Figure 1: Physical Systems *vs* Software Systems

Some entries in Table 1 require explanation. A conservative physical system is "history sensitive" because a perturbation affects the entire future history of the system. For example, if the moon is struck by an asteroid, its orbit will be permanently changed. In contrast, a dissipative system with a limit cycle returns to that cycle after a (suitably small) perturbation. For example, kicking a pendulum-driven clock may produce a temporary change in the motion of a pendulum, but eventually the pendulum will revert to its normal motion.

Most current software behaves like a conservative system, running in simple, unchanging loops. A change in the environment affects the program permanently, often fatally. The answer provided by the program tend to be either precisely correct or completely wrong, and the program provides no guidance as to which.

The software we would like to have behaves like a dissipative system. An environmental change causes a transient response, but the program eventually settles into a new equilibrium. The answers it proves are always reliable even though thy may not be precise. If the program cannot obtain an answer with confidence, it says so.

4   Advanced Programming Techniques

In this section, we review various approaches to programming that are more or less radical. We do not, for instance, mention the "conventional" approach: design and code.

Section 4.1 discusses techniques that are well-established but not always recognized as radical approaches to software development. Section 4.2 discusses techniques that have been in use for some time but have not yet (or only recently) been accepted as legitimate for software applications. Section 4.3 deals with software that we rather loosely refer to as "organic". Living organisms are robust; can we reverse engineer their design and use it in our own software?

4.1   Mainstream Methods

The methods reviewed in this section have already been tried with varying degrees of success. They are "mainstream" in that they are used in current software development.

4.1.1   Generators

Generators encapsulate knowledge about a particular application. Sort generators and report generators have been used since the earliest days of business programming. Compiler-compilers[3] also have a distinguished history. The theories of sorting and parsing are well understood. Consequently, it is possible to write a program that can generate a detailed and efficient procedure for sorting or parsing from relatively abstract requirements. So-called "visual languages" are modern examples of generators: they generate code for user interfaces from simple and intuitive input actions, hiding the low-level complexity of user-interface design. Some fourth-generation "languages" (4GLs) are COBOL generators; others generate code that is interpreted.

Generators encapsulate knowledge about a particular application. The theories of sorting and parsing, for example, are well understood. Consequently, it is possible to write a program that can generate a detailed and efficient procedure from relatively abstract requirements. User interface programming involves a large amount of detail and, usually, interaction with an extensive library of low-level functions. Interface requirements, however, follow relatively simple patterns:

In summary, generators can provide high levels of productivity for applications that are well understood.

4.1.2   Spreadsheets

Spreadsheets were introduced to perform quite specific tasks, generally involving accounting, but spreadsheets actually provide a new paradigm of computing. Although spreadsheets are implemented using conventional software techniques to run on conventional processors, we can view a spreadsheet as a virtual machine with many processors, each performing a computation that depends physically on its coordinates and logically on its role in the complete

---

[3]Now known, more accurately, as parser generators.

system. In fact, a spreadsheet can be viewed as a two-dimensional cellular automaton, and spreadsheets have been used to demonstrate aspects of automata theory (Toffoli and Margolus 1988).[4]

4.1.3 Layered Architectures

In a system structured in layers, "higher" layers invoke the services of "lower" layers. Layered architectures are not new (Corbato et al. 1965; Dijkstra 1968). There has been considerable interest in layered systems in which the lower layers are independent processes capable of making autonomous decisions (Albus 1981). These systems resemble biological systems in that they have a resilience of response that is lacking in conventional software.

A recent and unusual application of "layering" is the transaction monitoring system used by American Express: transactions are processed by a conventional software system, but an expert system acts as an "overseer", flagging unusual transactions for human attention.

Generalizing from this example, we can see that an advantage of the two-layer approach is that the normal task can be performed by simple and efficient code. Abnormal situations are detected and processed by a higher layer that may be much more complex than the base layer and may employ different techniques. This provides better separation of concern than the conventional approach, in which simple code is cluttered by rarely used exception handling code.

Brooks[5] has taken the layered approach a step further by introducing "subsumption architectures" (Brooks 1986). Lower layers are capable of independent simple behaviour. Higher layers provide more complex behaviour that "subsumes" the low-level behaviour (see Figure 2. In an emergency, such as the failure of a high-level component, the low-level layers can revert to their simple behaviour, maintaining the integrity of the system with reduced performance.
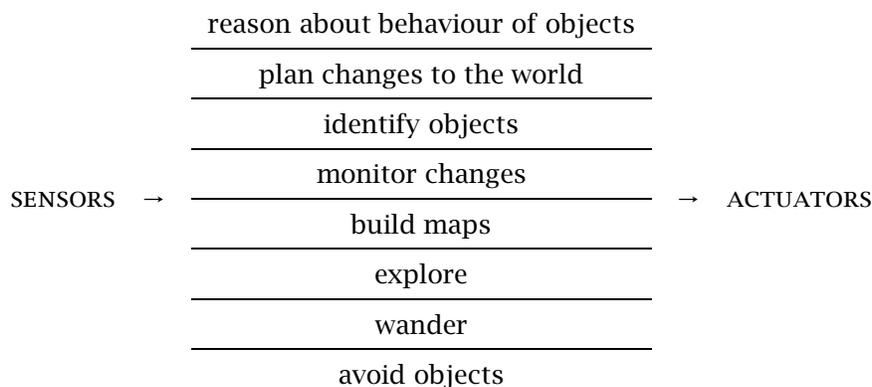
	reason about behaviour of objects	
	plan changes to the world	
	identify objects	
SENSORS →	monitor changes	→ ACTUATORS
	build maps	
	explore	
	wander	
	avoid objects	

Figure 2: Subsumption Architecture for a Robot (Levy 1992, page 279)

---

[4]Need a reference to the work on spreadsheets with animated images done at Xerox PARC.

[5]Rodney, not to be confused with Frederick.

4.2 Off the Beaten Track

In this section, we review programming techniques that have not yet been used for large-scale industrial development.[6]

4.2.1 Programs that Learn

(Langley and Simon 1995).

4.2.2 Reflective Programming

Adaptiveness is evident in *reflective programming* (Ferber 1989), where systems "know how they work" in a well-defined, technical sense.

4.2.3 Fuzzy Logic

Fuzzy logic[7] has been adopted with more enthusiasm by Japanese designers than by their western counterparts, and there are now a number of simple systems, such as domestic appliances, that rely on fuzzy logic.

Parnas pointed out the fragility of systems depending on classical logic, which distinguishes only "true" and "false". By the time that he did so, there was already considerable interest in the emerging discipline of "fuzzy logic", which replaces "true" and "false" by real numbers that are treated essentially as probabilities.

It is not clear whether fuzzy logic is actually a solution to the problems that we are addressing in this document. It may be easier and more intuitive to design a system based on fuzzy logic than a system based on classical logic, but the resulting system may still be fragile. Although probabilities are used, a small change to the system's inputs may still cause a large change in the outputs, for example, when a threshold is crossed.

4.2.4 Neural Networks

4.3 Organic Software

The growing field called (or mis-called) "artificial life" has shown that quite simple programs can have "organic" or "biological" properties (Kelly 1994). This is significant, because living organisms exhibit precisely the kind of adaptability and robustness that software needs. As yet, software engineering has disdained artificial life approaches. Perhaps it is time to take them seriously.

For example, computer simulations would allow the behaviours of a software system to be selected as survivors of adversarial testing. The current generation of computers has the power

---

[6]Some parts of this section are still under construction.
[7]Need a reference for fuzzy logic.

to perform this kind of processing. A framework or architecture would provide the environment in which the evolving system would function, thus giving it a wide range of potential behaviour. The building blocks of the emergent behaviour would probably be high-level, much higher then the conventional statements of a programming language. The following examples illustrate this idea.

4.3.1   Autonomous Agents

Networks have triggered interest in "agents" — programs that act autonomously (Maes 1995). The interesting feature of networks from our point of view is that software is replicated at nodes of the network. Consider, for example, a parameterized routing algorithm and assume that different nodes have different parameter settings. Each node could monitor the performance of its router and broadcast the results to other nodes. A node with an inferior router would adopt a better router from some other node. The key points here are that the software improves while it is performing useful work and that the software is evaluated under working conditions.

4.3.2   Evolutionary Development

We can improve a program by making changes to a single copy. If we have sufficient processing power, we can obtain improvements by allowing many versions of the program to evolve in a competitive environment. We can accelerate the "evolution" of the programs by introducing "parasites" that apply increasingly difficult tests to the evolving programs.

Thomas Ray's (1992) experiments with Tierra show that it is not hard to construct an environment in which evolution and adaptation can take place.

Hillis (1992) used an evolutionary approach to develop sorting networks in the Connection machine. The problem was to find a network for sorting 16 values (Knuth 1973, pages 227–29). Human attempts had moved from 65 nodes in 1962 to 60 nodes in 1969. Hillis's program found a 61 node network that is not quite as small as Green's 60 node network but is more resistant to small changes.

An important lesson of the "artificial life" experiments by Hillis, Ray, and others is that evolution is accelerated in the presence of parasites. This suggests a form of testing in which the software product evolves in competition with evolving tests. The resulting software would be less sharply optimized but more resilient to change.

4.3.3   Ant Simulations

An ant is a rather simple organism that displays complex behaviour because it exists in a complex environment. This observation (due to E. O. Wilson) has been experimentally confirmed by programs that simulate ant behaviour (Collins and Jefferson 1992).

Ant behaviour is of interest to software developers because it provides an example of a system that exhibits complex behaviour without itself being complex. An ant colony appears to function as an well-organized entity, although there is in fact no central control and the apparently organized behaviour is actually emergent.

4.3.4   Sparse Distributed Memory

Pentti Kanerva (1988) has described an algorithm for storage and retrieval called "sparse distributed memory" (SDM). A SDM has a large addressable space but a relatively small number of "hard" locations where data can be stored. For example, an address might have 1,000 bits, suggesting 2^{1000} locations, but only 1,000,000 ($\approx 2^{20}$) addresses correspond to physical locations.

SDM has interesting properties that arise from the binomial distribution of Hamming distance between addresses. Given an address that does not match a physical location, an iterative algorithm converges if the correspondence is close and diverges otherwise. The closeness threshold depends on the contents of the memory. SDM is in some ways more like human memory than computer "memory". A system with SDM, rather than a conventional database, for example, can display robust behaviour.

4.4   Summary

We are all familiar with the spectacular rate of advance in computer hardware: today's laptop has the power of yesterday's supercomputer. Improvements in hardware performance have been offset by the increasing flabbiness of software (Wirth 1995), but there are still a lot of spare cycles around.

Several of the ideas described in this section are, even by modern standards, inefficient. Some of them are outrageously inefficient. But this does not mean that we should rule them out for future systems. History shows that many software concepts that were once considered too inefficient to be practical have moved into the mainstream: high-level languages such as FORTRAN, LISP, and Smalltalk; recursion; structured programming; pixel-based screen management; and many others. Furthermore, when people realize the importance of a technique, they look for efficient ways of realizing it.

Consequently, we are not deterred by the apparent inefficiency of some of the methods outlined above. We may also propose, as part of this research, new inefficient ideas. If the ideas are useful enough, either the efficiency will be tolerated or acceptable solutions will be found.

Brooks (1978) suggests that our view of software development evolved from "writing" in the fifties, through "building" in the sixties, to "growing" in the seventies. The transition from the "waterfall model" (Royce 1970) to the "spiral model" (Boehm 1986) reflects the same evolution.

The importance of designing for change has long been recognized (Parnas 1979), although it has always been easier to proclaim than to practice. The mechanism of change has usually been understood to be external to the software itself. Software engineers make changes to an existing program, producing a new "version" that must be compiled, tested, and installed. Because software is so fragile, the alternative approach of making changes to systems while they are running has been practiced only rarely. The increasing use of computer networks has led to a shift in priorities: it is undesirable to bring down a network to enhance the software in a single node. Moreover, the network itself can be used to implement change in a distributed fashion. Consequently, there is an increasing need for software that can be enhanced with minimal interruption, perhaps even while it is running.

5   Towards Robust Software

In this section, we suggest some of the ways in which the ideas outlined above might be exploited in software design.

5.1   Fault Tolerant Software

Fault tolerance in software has been studied extensively. The following proposal, although it is very simple, shows how the ideas above (specifically, layered structure) might be used for fault tolerance.

Consider a table indexed by keys. Suppose that the keys are somewhat unreliable: for example, they might be people's names, and "Joe Jones" might (or might not) be the same person as "Joseph Jones". We could implement the table using two layers. The lower layer would provide a fast but stupid response, using perhaps exact match in a hash-table. The upper layer would monitor the lower layer, watching for potential errors (keys not found, close matches, etc.). When the upper layer notices a suspicious result, it blocks the output of the lower layer and uses a more sophisticated method, such as partial matching, to obtain a better solution. Whereas the lower layer is fixed, the upper layer has potential for learning to recognize ambiguous or incorrect keys.

Other applications of this principle come readily to mind. For example, a simple LR(1) grammar for a generator such as `yacc` becomes complex if we add error recovery to it. An alternative approach would be to use `yacc` as a lower layer, capable of detecting syntax errors, but passing such errors to a higher layer for deeper analysis.

The advantage of the two-layer approach is that the normal task is performed by simple and efficient code. Abnormal situations are detected and processed by different code that may be much more complex and may employ different techniques — rules, fuzzy logic, etc. This provides better separation of concern than the conventional approach, in which simple code is cluttered by rarely used exception handling code.

5.2   Adversarial Testing

One of the lessons of artificial life experiments is that evolution is accelerated in the presence of parasites. This suggests a form of testing in which the software product evolves in competition with evolving tests. The resulting software would be less sharply optimized but more resilient to change.

There are limits to this approach, at last with the present state of the art. For example, it is unlikely that instruction sequences would "evolve". A complex system, however, typically has many parameters that must be adjusted for good performance. Adversarial testing could be used to set the parameters.

5.3   Network Software

Networks provide a particularly interesting environment for our ideas, because software is replicated at nodes of the network. Consider, for example, a parameterized routing algorithm

and assume that different nodes have different parameter settings. Each node could monitor the performance of its router and broadcast the results to other nodes. A node with an inferior router would adopt a better router from some other node. The key points here are that the software improves while it is performing useful work and that the software is evaluated under working conditions.

5.4   Viruses

Viruses are normally considered to be bad news and best avoided, whether they affect our bodies or our computers. But viruses first appeared as worms — programs that tunnelled through networks looking for useful tasks to perform. A mutating virus could do a great deal of damage — fortunately, none have been sighted yet — but a mutating worm is potentially a useful and powerful form of software.

5.5   Frameworks

The concept "framework" arose in object oriented programming. Technically, a framework is a related collection of abstract classes that represents the "skeleton" of a system. To complete the system, "flesh" is provided in the form of concrete subclasses. One of the best known frameworks is the Model/View/Controller (MVC) framework of Smalltalk (Goldberg and Robson 1983).

Frameworks are important because they provide a practical means of reusing designs. Unlike function libraries, which typically provide canned low-level services, a framework defines the high-level organization of a system.

5.6   Patterns

Patterns were introduced by the architect Christopher Alexander (1979, 1977) and adopted for object oriented software development (Gamma, Helm, Johnson, and Vlissides 1995). Introducing his book, Alexander explains the role of a pattern in this way:

> Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it in the same way twice. (Alexander, Ishikawa, and Silverstein 1977, Page x).

Compared to other kinds of software artefact intended for reuse, patterns are presented at a high level of abstraction. They are not code, or even interface specifications. They are best understood as *recipes* from which an experienced designer or programmer can cook up a complete solution to a particular problem.

5.7 Visualization

Visualization is already an important technique for computer users, including mathematicians, physicists, and medical researchers (Gershon and Eick 1995). The use of visualization in software development is rudimentary: most CASE tools still work at the level of two-dimensional bubble-and-arc diagrams. There are a few exceptions, such as "tree maps" (Johnson and Shneiderman 1991) and the work that has been done at Xerox PARC (Robertson, Card, and MacKinlay 1993).

Before the invention of printing, people relied on their memories to a much greater extent than they do today. As Yates (1966) has shown, they developed elaborate techniques for remembering both stories and factual information. These techniques typically involved memorizing the layout of a complex structure such as a building; other facts were then memorized by associating them with features of the building. Modern computer games have a somewhat similar approach: players manoeuvre within a complex environment.[8]

The two "technologies" could be combined in a system that provides a view of a complex program as the interior of a building. Corridors in the building would represent interconnections between modules; labels on doors would define module interfaces; opening the door would expose the implementation of the module; and so on.

Note that visualization and virtual reality techniques are already being used for software *measurement*, mostly in the parallel processing domain (Pancake, Simmons, and Yan 1995).

5.8 Summary

The most obvious objection to these techniques that we have outlined is that many of them have been tried before without success. There have been numerous attempts to make computer programs behave like biological organisms, exhibiting resilience, adaptation, the ability to learn, or simply "intelligence". After more than fifty years of research (e.g. (McCulloch and Pitts 1943; Walter 1961)), most of these attempts have failed. If we are to continue, we must ask: what has changed?

Our view is that one significant change has precipitated many other changes and that these changes have led to important advances in our understanding. The one significant change is the enormous increase in the availability and power of computers. Every owner of a personal computer has access to more processing power than an advanced computing laboratory of the fifties or sixties. Furthermore, cycles are cheap: fast processors can be left running for hours or days. Abundance of processing power encourages experimentation, and experiments have yielded useful and often surprising insights.

One of the consequences of readily available computing power has been the development of the theory of complexity (Nicolis and Prigogine 1989). Mathematicians of the nineteenth century recognized that complexity existed.[9] The analytical methods that they possessed, however, were of little help in understanding the nature of the complexity.

---

[8]Others have pointed out connections between modern software and computer games: e.g. Vaskevitch (1995).

[9]For example, Cauchy knew that solving $z^3 = 1$ by Newton's method gave rise to complex basins of attraction (now known to be fractal).

Computer simulation has revealed many interesting phenomena. Now, perhaps, it is time for the computing disciplines themselves to reap some of the harvest that computers have created.

5.9   A Philosophical Note

It might seem that we are interested in the kind of software that adapts to the apparent needs of the user, or displays silly faces with useless suggestions. Nothing could be further from the truth. Our view is that the computer is a tool. As a tool, the computer should perform the tasks that we expect it to do rapidly, silently, correctly, and effectively. Ideally, the computer should be a transparent medium, in the sense that the mechanics of riding a bicycle or playing a guitar are transparent to skilled users. If you want a dumb companion, get a dog.

6   Conclusion

Software engineering is an inwardly focused discipline. The phases of software development (analysis, design, implementation, testing) were established thirty years ago and have not changed much since then. Design and implementation were, and are, based on algorithms and data structures. Only recently, as functional design has given way to object oriented design, have designers learned to work with software units larger than functions and procedures. Programming languages have evolved slowly to meet the needs of modularization and extensibility. Software engineering remains essentially reductionist: the whole is no more than the sum of the parts.

In contrast to software engineering, computer applications have developed with remarkable rapidity. To a large extent, the advances are related to improvements in hardware technology. Visualization, for example, requires fast processing, large memories, and high-resolution colour displays. Hardware advances have stimulated the development of new algorithms for applications such as image processing, digital signal processing, compression, and encryption. More importantly, the hardware advances have encouraged research into entirely new kinds of software. Dynamic systems theory and simulated evolution suggest that the whole may be greater than the sum of its parts.

References

Albus, J. S. (1981). *Brains, Behavior, and Robotics.* Byte Books (McGraw-Hill).

Alexander, C. (1979). *A Timeless Way of Building.* Oxford University Press.

Alexander, C., S. Ishikawa, and M. Silverstein (1977). *A Pattern Language*. Oxford University Press.

Boehm, B. W. (1986, March). A spiral model of software development and enhancement. In *Proc. International Workshop on the Software Process and Software Environments*, pp. 21–42. Reprinted in *ACM Software Engineering Notes*, 11:4 (August 1986) and in *IEEE Computer*, 21(5):61–72 (May 1988).

Brand, S. (1987). *The Media Lab: Inventing the Future at MIT*. Penguin Books.

Brooks, F. P. (1978). *The Mythical Man-Month*. Addison-Wesley. Anniversary Edition, 1995.

Brooks, F. P. (1987, April). No silver bullet: Essence and accidents for software engineering. *IEEE Computer 20*(4), 10–18.

Brooks, R. A. (1986, March). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 14–23.

Cohen, J. and I. Stewart (1994). *The Collapse of Chaos: Discovering Simplicity in a Complex World*. Penguin Books.

Collins, R. J. and D. R. Jefferson (1992). AntFarm: Towards simulated evolution. In (Langton, Taylor, Farmer, and Rasmussen 1992), pp. 579–601.

Corbato, F. J. et al. (1965). A new remote-accessed man-machine system. In *Proc. AFIPS 1965 FJCC*, pp. 185–247.

Dijkstra, E. W. (1968, May). The structure of the THE multiprogramming system. *Comm. ACM 11*(5), 341–346.

Ferber, J. (1989). Computational reflection in class based object oriented languages. In N. Meyrowitz (Ed.), *Proc. 16th ACM Symp. on Principles of Programming Languages*, pp. 317–326.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Gershon, N. and S. G. Eick (1995, November). Visualization's new tack: making sense of information. *IEEE Spectrum 32*(11), 38–56.

Glass, R. L. (1994, November). The software-research crisis. *IEEE Software 11*(6), 42–47.

Gleick, J. (1987). *Chaos: Making a New Science*. Viking.

Goldberg, A. and D. Robson (1983). *Smalltalk-80: The Language and its Implementation*. Addison-Wesley.

Hillis, W. D. (1992). Co-evolving parasites improve simulated evolution as an optimizing procedure. In (Langton, Taylor, Farmer, and Rasmussen 1992), pp. 313–324.

Johnson, B. and B. Shneiderman (1991). Tree maps: a space-filling approach to the visualization of hierarchical information structures. In *IEEE Visualization '91 Conference Proceedings*, pp. 284–291. IEEE Computer Society Press.

Kanerva, P. (1988). *Sparse Distributed Memory*. MIT Press.

Kelly, K. (1994). *Out of Control: the New Biology of Machines, Social Systems, and the Economic World*. Addison-Wesley.

Knuth, D. E. (1973). *Sorting and Searching.* Addison-Wesley. Volume 3 of *The Art of Computer Programming.*

Langley, P. and H. A. Simon (1995, November). Applications of machine learning and rule induction. *Comm. ACM 38*(11), 54–64.

Langton, C. G., C. Taylor, J. D. Farmer, and S. Rasmussen (Eds.) (1992). *Artificial Life II*, Volume X of *Santa Fe Institute: Studies in the Sciences of Complexity.* Addison-Wesley.

Leebaert, D. (Ed.) (1995). *The Future of Software.* MIT Press.

Levy, S. (1992). *Artificial Life: the Quest for a New Creation.* Pantheon Books.

Lewis, T. et al. (1995, August). Where is software headed? A virtual roundatable. *IEEE Computer 28*(8), 20–32.

Maes, P. (1995, November). Artificial life meets entertainment: Lifelike autonomous agents. *Comm. ACM 38*(11), 108–114.

McCulloch, W. S. (1965). *Embodiments of Mind.* MIT Press. Reprinted in 1988.

McCulloch, W. S. and W. H. Pitts (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics 5*, 115–133. Reprinted in (McCulloch 1965).

Miller, S. and H. Urey (1953). A production of amino acids under possible primitive conditions. *Science 117*, 528–9.

Nicolis, G. and I. Prigogine (1989). *Exploring Complexity: an Introduction.* W.H. Freeman.

Pancake, C. M., M. L. Simmons, and J. C. Yan (1995, November). Guest editors' introduction: Performance evaluation tools for parallel and distributed systems. *IEEE Computer 28*(11), 16–19. See also other articles in the same issue.

Parnas, D. L. (1979, March). Designing software for ease of extension and contraction. *IEEE Trans. Software Engineering 5*(2), 128–138.

Parnas, D. L. (1985, December). Software aspects of strategic defense systems. *Comm. ACM 28*(12), 1326–1335.

Peitgen, H.-O. and D. Saupe (1988). *The Science of Fractal Images.* Springer-Verlag.

Ray, T. S. (1992). An approach to the synthesis of life. In (Langton, Taylor, Farmer, and Rasmussen 1992), pp. 371–408.

Robertson, G. G., S. K. Card, and J. D. MacKinlay (1993, April). Information visualization using 3D interactive animation. *Comm. ACM 36*(4), 56–71.

Royce, W. W. (1970). Managing the development of large software systems: Concept and techniques. In *1970 WESCON Technical Papers, Western Electric Show and Convention*, pp. A/1-1–A/1-9. Reprinted in *Proceedings of the 11th International Conference on Software Engineering*, Pittsburgh, May 1989, pp. 328–338.

Srinivas, M. and L. M. Patnaik (1994, June). Genetic algorithms: a survey. *IEEE Computer 27*(6), 17–26.

Toffoli, T. and N. Margolus (1988). *Cellular Automata Machines: a new environment for modeling.* MIT Press.

Vaskevitch, D. (1995). Is any of this relevant? In (Leebaert 1995). MIT Press.

Walter, W. G. (1961). *The Living Brain.* Penguin Books. First published by Duckworth, 1953.

Wirth, N. (1995, February). A plea for lean software. *IEEE Computer 28*(2), 64–68.

Yates, F. A. (1966). *The Art of Memory*. Routledge & Kegan Paul. Republished by Penguin Books, 1968.