Engineering Artificial Intelligence Software

Derek Partridge

Engineering Artificial Intelligence Software

Derek Partridge
Department of Computer Science
University of Exeter

intellect
Oxford, England

First published in Great Britain in 1992 by
Intellect Books
Suite 2, 108/110 London Road, Oxford OX3 9AW

First published in the US in 1992 by
Ablex Publishing Corporation
355 Chestnut Street
Norwood, New Jersey 07648

Consulting editor: Masoud Yazdani
Copy editor: Pamela Clements
Cover design: Mark Lewis

British Library Cataloguing in Publication

Partridge, D. (Derek) 1945-
Engineering Artificial Intelligence Software
1. Computer systems. Software. Design. Applications of artificial intelligence
I. Title
005.12

ISBN 1-871516-06-4

Library of Congress Cataloging-in-Publication Data

Partridge, Derek. 1945-
Engineering Artificial Intelligence Software / by Derek Partridge.
p. cm.
Includes bibliographical references and index.
ISBN 0-89391-778-8
1. Artificial Intelligence--Software. I. Title.
0336.P37
006.3'0285'5--cc20 90-26630
CIP

Printed and bound in Great Britain by Billing & Sons Ltd, Worcester

Engineering Artificial Intelligence Software

Preface

This book is aimed at the computer-literate person who wishes to find out about the reality of exploiting the promise of artificial intelligence in practical, maintainable software systems. It cuts through the hype, so commonly associated with discussions of artificial intelligence, and presents the realities, both the promise and the problems, the current state of the art, and future directions.

It is not another expert systems book. Expert systems are viewed as just one manifestation of AI in practical software; the issues raised in this book are much broader. Expert systems are discussed, but as a source of lessons about the dangers as well as the beneficial possibilities for adding AI to practical software systems.

In the opening three chapters, we take a long hard look at the conventional wisdom concerning software engineering - what the goals are, the rationale behind them, and the realities of the situation. This is done in part because efforts to engineer AI-software appear to undermine the accepted foundations of conventional software engineering so we need to determine how solid the foundations really are; it is also done because in attempting to engineer AI-software we subject the standard procedures of software design and development to close scrutiny - our attempts to build robust and reliable AI-software provides a magnifying glass on the conventional procedures.

Chapter 4 elaborates on the prototyping schemes described in Chapter 3 and uses this well-accepted methodological strategy to take us on into the more contentious domain of evolutionary and exploratory software design and development. This move places us squarely within the general paradigm (i.e. incremental system development) from whence an effective model for engineering AI software will emerge. This chapter concludes with a presentation of the conventional paradigms for software development which sets the scene for the `new paradigms' which constitute the next chapter.

Chapters 1 to 4 are somewhat polemical in nature, unashamedly so. Is this appropriate in a text book? Clearly, I believe that it can be, and it is in this case. As attractive as it might be to provide an unbiased presentation of how to build AI software, it is just not possible. How best to build AI software, and even whether it is a reasonable endeavour to embark on in the first place, are controversial topics. I have attempted to present the major competing alternatives whenever possible, and I haven't tried too hard to hide my personal preferences. I don't think that

this style of writing is necessarily out of place in a text book. In fact, I think that it is sorely needed in this particular subject area. The prevalence of uncritical texts is largely responsible for the current state of passive acceptance of what should really be hotly debated issues, e.g. that software construction should mirror conventional engineering practice, or that the single key component is an abstract specification which can be complete.

I clearly don't have an exclusive insight into the problems that permeate the software world; many before me have seriously questioned elements of the conventional wisdom and offered a possible remedy. In Chapter 5 I present just a few of the new paradigms that have been proposed in recent years. This serves to open up the problem, to emphasize the particularly problematic elements, and to reassure you that there is no quick fix that I have overlooked.

In Chapter 6 I refocus the narrative and examine specific components of an effective incremental software design and development paradigm - a discipline of exploratory programming. None of these component features appear to be readily available, and some seem to offer little hope of availability, ready or otherwise. One of the intentions of this chapter is to show that despite the actual difficulty of realizing these essential component features, the appearance of impossibility is only an unfortunate illusion.

The next two chapters present two, very different, sources of reason for guarded optimism for the engineering of AI software. Chapter 7, on Machine Learning, reviews several facets of this complex AI subfield that have had an impact on practical software development. Chapter 8 comprises the main concession to expert systems' technology. As I said at the very beginning, this is not an expert systems book, but this new technology cannot be ignored in a book that deals with the design and development of AI software. However, we do not retell the famous exploits of the star performers in this field, nor do we examine particular mechanisms used to achieve expert-level performances. What we do is to look at how, in general terms, these celebrated AI-software systems were developed, and equally important why so many of them have failed to surmount the prototype stage. The lessons are both positive and negative, and it may be the negative ones that are the most instructive.

The penultimate chapter attempts to draw together many of the lines of reasoning developed earlier. It attempts to organize these threads of argument into a more coherent whole - the umbrella of software support environments. To conclude this chapter two other seemingly desirable styles of approach to the problems of engineering AI software are examined.

In the final chapter, Chapter 10, we bring in the `societal' aspects of the general problem. It is people that build software systems (and that may be **the** problem, the advocate of automatic programming might be tempted to interject), and for any significant system it is definitely people rather than a single person. Just as software systems are not built in a personless vacuum, they are seldom used in one either. It is all too easy for the `technician' to focus exclusively on the technical problems and forget that many people are in fact bound up with the problem, and in many different ways. So this last chapter, in addition to providing a

summation of all that has gone before, briefly reviews these `societal' problems of software development. For they cannot be temporarily shelved to be tackled after the technical ones have been solved: they are part and parcel of the technical problems of engineering AI software - indeed, of engineering all large-scale software. And to neglect the people aspect may leave us attempting to solve fictitious, purely technical problems.

Finally, the embryonic state of the art in the engineering of AI-software (despite what you might hear to the contrary) means that this is not, and cannot at the present time be, a `manual' to guide the interested reader through a detailed procedure for constructing robust and reliable AI-software products. Although I do present and discuss specific systems (even a few commercially available systems) whenever possible, the book is by no means saturated with expositions of the shells, tools or environments that you can just go out and buy in order to get on with engineering some AI-software. What you will find (I hope) is a comprehensive and coherent examination of the many problems that engineering AI-software involves, as well as a consideration of the alternative routes to solution of these problems. This book is certainly not the last word on this important subject, but it may be a good start.

Acknowledgements

As with all substantial pieces of work the author cannot take 100 per cent of the credit, only of the blame. I would like to thank several anonymous reviewers who helped curb my excesses and Sue Charles who drew most of the figures.

Derek Partridge

Contents

List of figures

1
Introduction to Computer Software

1.1 Computers and software systems

Software systems are programs, usually large ones, running on a computer. Despite several decades of concerted effort, the design, implementation, and maintenance of such systems is more of an art than a science. That is to say, the development and maintenance of such systems are processes dominated by loose guidelines, heuristic principles and inspirational guesswork, rather than formally defined principles and well-defined techniques.

The advent of electronic digital computers and the subsequent, almost miraculous, advances in electronic circuitry technology have provided mankind with an amazing tool. It has taken humanity into a new age. Electronic computers have opened up possibilities that were hardly dreamed of just a few decades ago. We have rockets that can be accurately guided to the outermost planets and can then take pictures to send back to us. We have a banking system, involving plastic cards and remote automatic tellers, that provides us with astounding financial flexibility, and so on.

In sum, computers have moved us into domains of unbelievable complexity, and enable us to manage them fairly successfully - most of the time. In fact computers don't just enable us to deal with situations of far greater complexity than we could possibly manage without them, they positively lure us into these domains of excessive complexity. The modern computer is an exceedingly seductive device: it tempts us with the promise of its great power, but it also entices the unwary to overstep the bounds of manageable complexity.

Computer systems designers are well aware of this danger, and that small specialist sector of society whose role is to construct software systems has laboured to produce a framework from which reliable, robust, and maintainable, in a phrase, practically useful software systems are likely to be forthcoming. This framework is the central feature of the discipline of **software engineering**. Observance of the strictures of software engineering *can* lead to the production of high-quality software systems, but there are no guarantees. Software system design and construction is thus a skilled art (i.e. a blend of artistic flair and technical skill), but then so is much of science in all domains, despite the widespread, naive views to the contrary. So what exactly is software engineering?

1.2 An introduction to software engineering

What is software engineering? Well according to one source:

> Software engineering is the introduction of formal engineering principles to the creation and production of software. A scientific or logical approach replaces the perhaps more traditional unstructured (or artistic) methods.
>
> DTI, *Software Engineering Newsletter*, Issue No. 7,1988

This definition is on the right track, but is perhaps more a definition of some future desired situation than the current reality. And clearly I have some reservations about the 'scientific' aspirations explicitly mentioned in the definition. I don't really know what this word means, but I suspect that it is being (mis)used as a synonym for 'logical'. A further point of contention that will emerge later when we come to a consideration of the promise and problems of artificial intelligence (AI) in practical software systems is that the desire for 'a scientific or logical approach' may be born of a fundamental misconception, and one that AI illuminates.

Ince (1989) presents an altogether less slick but far more realistic characterization of the essence of software engineering.

> Software engineering is no different from conventional engineering disciplines: a software product has to meet cost constraints; the software engineer relies on results from computer science to carry out system development; a software system has to carry out certain functions, for example in a plant monitoring system those of communicating pressure and temperature readings to plant operators; and a software developer has to work under sets of constraints such as budget, project duration, system response time and program size.
>
> Ince (1989) p. 4

This definition (or description) again tries to account for the 'engineering' part of the label, but it avoids the mythological associations of the term instead of stressing them as in the previous definition. Engineering is not an exact science; it is not a discipline characterized by formal techniques; the 'logical approach' (under any formal interpretation of 'logical') has no major role in most types of engineering. In fact, much engineering is saturated with rule-of-thumb procedures which experience has shown will mostly work, i.e. it is just like much of the practice of software system building. Precise calculation and use of formal notations certainly have a role in engineering (as Parnas, 1989, for example stresses) and they also have a role in software engineering. The unanswered questions are: do they have central roles and do they have similar roles within these two disciplines?

Yet, it is undeniable that software systems crash with amazing regularity whereas bridges and buildings very seldom fail to perform adequately over long periods of time. This is the crucial difference between these two disciplines that leads us to think that software builders have useful lessons to learn from bridge builders, i.e. real

engineers. But coining the name 'software engineering' and yet setting a course for the realms of the logicist does not seem to be a rational way to capitalize on the success of real engineers.

Can we reasonably expect to be able to exploit engineering know-how? Are software systems like bridges, or buildings, or complex machinery? Or is the depth of similarity merely wafer thin? These are difficult questions and, moreover, ones that will intrude on our deliberations quite starkly when we bring AI into the picture. So, before launching into an investigation of software life cycles, I'll indulge in a minor diversion which will be seen to pay off later on in this text.

1.3 Bridges and buildings versus software systems

Let me tabulate a few differences and similarities between these two sorts of artefacts. First some similarities:

1. they are both artefacts, i.e. man-made objects;
2. they are both complex objects, i.e. have many interrelated components;
3. they are both pragmatic objects, i.e. are destined for actual use in the real world - contrast works of art;

but there are, of course, some significant differences.

bridges and buildings	**software systems**
concrete objects	formal-abstraction-based objects
non-linguistic objects	linguistic objects
non-malleable objects	malleable objects
simple or relatively unconstrained functionality	precise, complex functionality

Some explanation of this tabulation is required, I suspect. There are clearly important differences between an elaborate pile of reinforced concrete and steel (say, a bridge or building) and a computer program (i.e. a software system). The former is a solid physical object which is typically difficult and expensive to change. If, for example, you're curious about the ramifications of two span-supports rather than three, or about the implications of doubling the thickness of a steel girder, the chances of being able to satisfy your curiosity on the engineered product are very slim. The engineer must resort to modelling (i.e. constructing small-scale models, rather than mathematical models - although this is done as well) to answer these sorts of questions, or verify predictions of this nature, as they so often do. Notice that such modelling is an alternative to formal analysis, and not the preferred alternative - it's more expensive, slower, and less accurate. At least, it would have all of these less desirable characteristics *if formal analysis were possible*, but it usually isn't except in terms of gross approximations (the mathematical models) that need to be supported by empirical evidence such as model building

can provide.

So, because of the fact that engineering is not a formal science, the notion of modelling, of building prototypes, is heavily exploited. Take note of the fact that typically the model (or prototype) and the final engineered product are totally different objects, and they have to be because the impetus for this type of modelling is to be found in the nature of the non-malleability of the engineered final product.

Software-engineered products (i.e. programs) are quite different with respect to malleability. Programs are very easily alterable. Tentative changes typically cost very little in time, money, or effort. In fact, programs are much too easily alterable; in my terminology, they are highly malleable objects. Thus it would seem that the need to build models in order to explore conjectures, or verify deductions, is absent. Yet modelling, in the sense of building prototypes, is common practice in software engineering. "Have the software engineers misunderstood the nature of their task?" we ask ourselves. I'm very tempted to answer "yes" to this question, but not with respect to the need for prototype model building. Why is this?

First, notice that a model of a software system *is itself* a software system. This fundamental difference between engineering and software engineering leads to the conclusion that seemingly similar processes (i.e. prototype model building) may serve rather different purposes within these two classes of engineering. At the very least, it should serve as a warning against over-eagerness in equating the activities of real engineers and software engineers, even when they seem to be doing quite similar things.

An important difference between the products of engineering and of software engineering stems from the precise nature of the abstract medium used to build programs - it is a linguistic medium. A software system is a statement in a language. It is not a statement in a natural language, and it may appear quite alien to the average linguistically competent, but computationally illiterate, observer. Nevertheless, a software system is a linguistic statement, and although natural languages and formal languages have much less in common than their names would suggest (formal languages: another readily misinterpretable label), there are some significant similarities. One consequence of this feature of software engineered products is that we may be tempted to entertain the view that programs might also be theories - we don't typically spare much thought for the suggestion that bridges might be theories (in structural mechanics?) or even that they might contain their designs (the blueprints are in there somewhere!). Similarly, software engineers are typically not tempted too much to view their artefacts as theories, but, when we later let AI intrude into these deliberations, we'll see that this odd notion gathers considerable support in some quarters.

A most important consequence of the fact that programs are composed of formal linguistic structures - i.e. programs are formal statements- is that they therefore invite being mauled with mechanisms that can boast the unassailable attribute of mathematical rigour. In particular, they are constantly subjected to logical massage, of one sort or another - especially the sorts that hold a promise 'proof of correctness.'

The last-listed point of difference concerns the functionality of

acceptable objects - i.e. how they behave in the world. You might be tempted to observe that many engineered objects don't behave, they just are. The function of a bridge is purely passive: it doesn't act *on* the world, it just sits there and allows things to run over it. So it does have this passive functionality (which is a point of difference with software systems), but more importantly the functionality is relatively simple - crudely put, its function is to sit there without breaking or cracking and let things run over it (not much of an existence I admit, but that's about the extent of it if you're a bridge).

"Ah, but certain engineered objects, such as steam engines, do have a complex, active functionality just like software systems." you might respond. "Yes and no," I would reply. They do have complex, active functionality, but it's very different from the functionality of software systems in several crucial respects. It's less tightly constrained and it's much more modular - i.e. engineered products have relatively broad ranges of acceptable functionality, and the overall functionality can be decomposed into relatively independent components. Thus, a building must exhibit complex functionality, both active and passive, but to a good first approximation the adequacy of the doors, doorways, corridors and stairs in supporting movement throughout the structure is independent of the adequacy of the central heating system, and of the performance of the toilets, and of the performance of the roof, etc. The undoubted complex functionality can be broken down, *to a reasonable approximation*, into relatively independent functional components - this results in a reduction in effective complexity. And, although the doors, doorways, etc. must conform to building codes, almost any reasonably large hole in the wall will operate adequately as a door: something pretty drastic has to happen before it will fail to support the necessary function of providing access from one space to another. Software systems are rather different. We strive for modular functionality and achieve it to varying degrees dependent upon the nature of the problem as well as the skill of the software engineer. But the medium of programming lends itself to a hidden and subtle interaction between components that is not possible (and therefore not a problem) when building with nuts, bolts, concrete, steel, plastic, etc. There is also a positive inducement to employ functional cross-links: they can buy software power cheaply in terms of program size and running time (the important issue of software power is considered fully later).

Finally, there are usually tight constraints on what functionality is acceptable: software systems typically operate under precisely defined functional constraints, if the actual functionality strays from the tight demands of the system specification then it is likely to be inadequate, incorrect, even useless in its designated role. Notice that not all (or even most) of these constraints are explicitly stated; many tight functional requirements become necessary in order to realize the specified requirements given the particular implementation strategy selected. They just emerge as the software engineer strives to develop a correct computational procedure within the web of constraints on the task. These are the implicit functional constraints on the particular software system; they have to be actively searched out and checked for accuracy.

In his polemic on what should be taught in computer science, Dijkstra

makes the preliminary point that modern computer technology introduces two "radical novelties". The particularly relevant one here "is that the automatic computer is our first large-scale digital device" and consequently "it has, unavoidably, the uncomfortable property that the smallest possible perturbations - i.e. changes of a single bit - can have the most drastic consequences." Whereas most engineered artefacts "are viewed as analogue devices whose behaviour is, over a large range, a continuous function of all parameters involved" (Dijkstra, 1989, p. 1400).

In sum, there are a few similarities between bridges and software systems, but there are many salient differences. So why this general exhortation to try to build and design programs in the mould successfully used for bridges? Is it a misguided campaign fuelled by little more than desperation? What the software people really hope to import into their discipline is product reliability. But are bridges, buildings, and steam engines reliable because the technology is well understood, in the sense of well defined, or because the artefacts are produced as a result of rigid adherence to a complete and precise specification of desired behaviour? The answer is not clearly 'Yes'. And yet we find many advocates of "formal methods" in the world of software design and development who appear to subscribe to something like this view. The term "formal methods", when used in the software world, focuses attention on the notions of abstract formalized specifications for software, and provable correctness of a program with respect to such a specification. Gibbins (1990) discusses the question "What are formal methods?" and quotes with approval from an Alvey Software Engineering Strategy Document:

> "A formal method is a set of rigorous engineering practices which are generally based on formal systems and which are applied to the development of engineering products such as software or hardware". (p. 278)

Here we see again an attempt to roll formal techniques and engineering practices and products all into one ball, as if it's indisputable that they meld together well. One question we really ought to ask ourselves is, 'do they?' Is the use of formal techniques the key to engineering reliability? Or could it be that well-engineered products are reliable because of a long tradition that has allowed the heuristics to be refined? More importantly, are they reliable because their relative functional simplicity permits the incorporation of redundancy in a similarly simple way? If you want to be sure that the bridge will function adequately, then, when you've finished the crude calculations, you double the size of the spanning girders, pour more concrete into the supporting pillars, etc. What you don't do is set about trying to prove that your calculations will guarantee adequate functioning. The functional complexity of software systems seems to make them unamenable to a similar coarse-grained style of adding redundancy to safeguard performance requirements - i.e. you can't just double the number of statements in a program, or beef up the supporting procedures by adding more of the same statements. So, while the engineering paradigm in general (i.e. formal approximation bolstered by added redundancy and a wealth of experience) may be appropriate, it can be misguided to look too closely and uncritically at the details of engineering practice, and even worse to aspire to a fiction of supposed