INTERNATIONAL COMPUTER SCIENCE SERIES

SOFTWARE PROTOTYPING FORMAL METHODS AND VDM

Sharam Hekmatpour
Darrel Ince

SOFTWARE PROTOTYPING, FORMAL METHODS AND VDM

Sharam Hekmatpour
University of Melbourne

Darrel Ince
The Open University, Milton Keynes

ADDISON-WESLEY
PUBLISHING
COMPANY

Wokingham, England · Reading, Massachusetts · Menlo Park, California
New York · Don Mills, Ontario · Amsterdam · Bonn
Sydney · Singapore · Tokyo · Madrid · San Juan

Cover designed by Crayon Design of Henley-on-Thames and
printed by The Riverside Printing Co. (Reading) Ltd.
Printed in Great Britain by The Bath Press, Avon.

First printed 1988.

British Library Cataloguing in Publication Data
Hekmatpour, Sharam
Software prototyping, formal methods and
V.D.M. — (International computer science
series).
1. Computer systems. Software. Development.
Prototyping
I. Title II. Ince, D. (Darrel) III. Series
005.1

ISBN 0–201–17572–X

Library of Congress Cataloguing in Publication Data
Hekmatpour, S. (Sharam), 1961–
Software prototyping, formal methods, and VDM/Sharam Hekmatpour
and Darrel C. Ince.
p. cm. — (International computer science series)
Bibliography: p.
Includes index.
ISBN 0–201–17572–X
1. Computer software — Development. I. Ince, D. (Darrel)
II. Title. III. Series.
QA76.76.D47H45 1988
005.1′1—dc19 88-19400
CIP

SOFTWARE PROTOTYPING, FORMAL METHODS AND VDM

Preface

Just a few years ago, rapid prototyping was an alien subject to software developers. Today, there is hardly anyone in the software community who has not at least heard of the term. Having recognized the importance of the concept, an increasing number of software houses are now actively engaged in setting up working groups on prototyping, and most international conferences on software engineering and Human–Computer Interface are devoting entire sessions to the subject. Although general interest in this area has risen dramatically and research has been intensified, there is still a serious shortage of material on the subject as well as software tools to support it.

This book aims to alleviate some of these shortcomings. It first introduces the reader to rapid prototyping by examining the state of the art, and then describes a prototyping methodology based on formal methods of software development. The latter is supported by a wide-spectrum language, embedded in a UNIX-based prototyping environment called EPROS, which enables very rapid generation of working prototypes from a formal description of a system.

The book is intended for four classes of readers: researchers in software engineering, developers who use formal methods of software development, industrial staff who are looking for viable prototyping techniques, and university lecturers who are interested in using a software tool in their formal methods and prototyping courses. Apart from the first three chapters, which are of introductory nature, the rest of the book assumes that the reader is familiar with the Vienna Development Method (VDM) formal specification notation. No attempts have been made to teach VDM in detail, since there are already two excellent introductory books on the subject [Jones 1980a, Jones 1986]. We have also assumed that the reader is familiar with the notion of programming in general and a high level programming language like C or Pascal in particular.

The organization of this book is as follows. Chapter 1 provides some motivation and explains why prototyping is important. Chapter 2 discusses prototyping in more depth and provides a categorization of approaches to prototyping. Chapter 3 describes a number of prototyping techniques in some detail. Chapter 4 gives an overview of the EPROS prototyping system, its development procedure and the wide-spectrum language it is based on, and describes the available facilities in some detail. The use of the specification notation is illustrated by a case study in Chapter 6. Chapters 7, 8 and 9 describe the implementation, user interface development, and meta abstraction facilities of EPROS, respectively. The final chapter describes a second case study which puts the described techniques into practice. The last version of the prototype developed in this chapter may be found in Appendix B. Appendix A serves as a reference manual for EPROS, and describes the formal syntax of its notation, its libraries, and its command language.

S. Hekmatpour and D. Ince
June 1988

Acknowledgements

The authors and publishers would like to thank the following:

- Association for Computing Machinery for permission to quote from McCracken, D.D. and Jackson, M.A. 1982. 'Life cycle concept considered harmful,' *ACM SIGSOFT Software Engineering Notes*, **7**(2) pp. 29–32.
- Elsevier Science Publishers for permission to quote from Bally, L., Brittan, J. and Wagner, K.H. 1977. 'A prototype approach to information system design and development,' *Information and Management*, **1** pp. 21–26.
- Oxford University Press for permission to reproduce material written by the authors and previously published in Zorkoczy, P.I. (ed.) 1986. 'Rapid software prototyping,' *Oxford Surveys in Information Technology,* **3** pp. 37–76.

A note on language
For reasons of simplicity, the pronoun 'he' is used to relate to both male and female throughout the book.

Contents

Chapter 1
Introduction

This chapter briefly describes some of the problems encountered with those conventional software development techniques associated with a phased software life cycle, and outlines how a prototyping approach is capable of overcoming these problems.

1.1 The life cycle model

For the past twenty years or so, software system development has been based on a model, commonly referred to as the software **life cycle** model [Zelkowitz *et al.* 1979, Boehm 1981, Sommerville 1982, Shooman 1982, Fox 1982]. Though characterized differently by different authors, its overall theme is well understood and universally acknowledged. The life cycle model leads to a software development strategy which is usually called the phase-oriented, the linear or the traditional strategy.

The life cycle model essentially advocates that software projects should consist of a number of distinct stages. These are: requirements analysis, requirements specification, design, implementation, validation, verification, operation and maintenance. **Requirements analysis** is concerned with deriving, from the customer, the desired properties and capabilities of a proposed software system. **Requirements specification** involves stating the system functions and constraints in a precise and unambiguous way. **Design** is the task of producing, and consequently refining solutions that satisfy the specification. **Implementation** is the act of realizing the design in a programming language which can be executed on the target machine. **Validation** is the process of checking that a system fulfills its user requirements. **Verification** has the objective of ensuring that the end product of each of the first four stages matches its input. **Operation** is the activity of installing and running a completed system in its intended environment. Lastly, **maintenance** is the process of modifying a system, during its operational lifetime, to correct detected errors, improve performance, and incorporate newly emerging requirements.

The life cycle model was originally derived from the hardware production model of requirements, fabrication, test, operation and maintenance [Blum 1982]. It primarily reflects management concerns in production, such as planning, control, budget expenditure and resource allocation. Its aim is to provide a basis for estimating the correct distribution of labour and capital over a well planned period of time by dividing the production process into a number of rationalized phases, each with its own milestones and deliverables.

Central to the model is its linear structure; with exception of validation and verification, all other stages are carried out linearly, i.e., each stage begins only when the previous stage has been completed. The model works very well in hardware production; its appropriateness for software development, however, is becoming increasingly questionable.

1.2 Deficiencies of the life cycle

Software producers who currently use the life cycle model have to cope with three unpleasant facts: (i) the earlier an activity occurs in a project the poorer are the notations used for that activity, (ii) the earlier an activity occurs in a project the less we understand about the nature of that activity, and (iii) the earlier an error is made in a project the more catastrophic the effects of that error. For example, early requirements and specification errors have typically cost a hundred to a thousand times as much as those made during implementation [Boehm 1981], and have lead to a number of multi-million dollar projects being cancelled.

Increasing user dissatisfaction with software since the early nineteen seventies has motivated researchers to pay greater attention to the earlier stages of software development [Ramamoorthy *et al.* 1984]. As a result, many requirements analysis and specification techniques have been invented [Davis and Vick 1977, Ross and Schoman 1977, Taggart and Tharp 1977, Levene and Mullery 1982, Lehman and Yavneh 1985], some of which are even computerized [Smith and Knuth 1976, Teichroew and Hershey 1977, Bell *et al.* 1977]. At the same time there is a rapidly increasing interest in formal, more mathematical methods of software development which adherents claim lead to more reliable systems which have an increased probability of meeting user needs [Musser 1979, Davis 1979, Jones 1980b, Silverberg 1981].

Unfortunately, even when a software developer uses modern notations and techniques, success is likely only when the application is both well understood and supported by previous experience [Bally *et al.* 1977, Blum and Houghton 1982, Brittan 1980]. The current rate of growth in hardware has meant that each year large numbers of new applications emerge for which the old knowledge is inadequate. Faster and larger, cheaper memories mean that computers are being used in novel projects where the relation of the computer to its environment, to human operators, and to other computers has not been researched adequately. Many such projects are based on specifications which are not true reflections of the customer's requirements. This is due to three reasons.

First, there is usually a significant cultural gap between the customer and the developer and the way they communicate [Christensen and Kreplin 1984]. Consequently, a customer often finds it extremely hard to visualize a system by simply reading a technical system specification document [Gomaa and Scott 1981, Mayr *et al.* 1984]. If the customer is unable to visualize such a system then validation during the early part of the project becomes a very error prone activity. Indeed, the difficulties involved in communication with the user can be a serious barrier to proper development [McCracken and Jackson 1982]:

> 'The life cycle concept perpetuates our failure so far, as an industry, to build an effective bridge across the communication gap between end-user and system analyst. In many ways it constraints future thinking to fit the mold created in response to failures of the past.'

Second, the customer, unfamiliar with information technology, may have produced very vague requirements which could be interpreted arbitrarily by the developer [Brittan 1980]. Third, empirical evidence [Ackford 1967] suggests that once a user starts employing a computer system, many changes occur in his perception as to what the intended system should do; this obviously invalidates the original requirements. As a result, user requirements are often a moving target, and producing a system that meets them is a risky and error prone activity.

A further complication is that a software project of considerable size may take many years to complete; during this time the user requirements, as well as the user environment, may change considerably, making the final system even more obsolete [McLean 1976, Gladden 1982, Ramamoorthy *et al.* 1986]. This is graphically described by [Blum 1982]: 'Development is like talking to a distant star; by the time you receive the answer, you may have forgotten the question.'

The life cycle model is strongly based on the assumption that a complete, concise and consistent specification of a proposed system can be produced prior to design and implementation. The validity of this assumption has been challenged and refuted by a number of authors [Swartout and Balzer 1982, McCracken and

Jackson 1982, Shaw 1985]. In many cases a complete specification cannot be produced, simply because the user does not really know what he wants [Berrisford and Wetherbe 1979, Parnas and Clements 1986].

Lack of experience in projects where it is almost impossible to construct a precise specification leads to the situation where the customer requirements can be established only when a complete software system has been built and when the system can be examined in a fully concrete form [Blum and Houghton 1982]. For this reason many systems end up being written at least twice. To quote Brooks [1975]: 'Plan to throw one away; you will, anyhow.'

There are numerous examples in the literature of substantial modifications of systems during maintenance because of inadequate requirements analysis. For example, it has been reported [Boehm 1974] that in some large systems up to 95% of the code had had to be rewritten to meet user requirements. Even more formal, improved techniques and notations for requirements specification are not helpful in this respect, as the transition from the user conceptual model of a system to a specification of the system is an inherently informal process [Leibrandt and Schnupp 1984].

All evidence, therefore, suggests that the life cycle model has many shortcomings which may have adverse effects on software projects. This is, of course, not to say that this model should be rejected outright. To the contrary, in certain areas, such as embedded software and real time control systems, it is the most rational approach and indeed the best way of controlling the complexity of such projects. However, for the majority of other applications, especially those related to commercial data processing, it is inappropriate and has many deficiencies which are too serious to be ignored. The deficiencies may be summarized as follows.

- It is unable to cope with vague and incomplete user requirements [Brittan 1980, MacEwen 1982].
- It discourages feedback to the earlier stages because of the cost escalation problems [Bastani 1985].
- It cannot predict the effects of introducing a new system into an organization before the system is complete [Keen 1981].
- It cannot properly study and take into account the human factors involved in using the system.
- It introduces a computer system into an organization suddenly. This is a rather risky approach since users are known to resist significant, sudden social changes [Rzevski 1984].
- The customer may have to wait for a long time before actually having a system available to him for use. This could have undesirable effects on customer trust and may cause frustration [Gladden 1982].
- The final product will, at best, reflect the user requirements at the start of the project and not the end. In long projects, these two may differ considerably

due to changes in the customer's organization and practices.

- Once the users start employing the final system and learn more about it, their views and intentions change significantly. Such changes in user perception can by no means be predicted [Clark *et al.* 1984].

1.3 The prototyping solution

In the light of the difficulties described above, many researchers have arrived at the conclusion that software development, particularly during its early stages, should be regarded as a *learning process* and practised as such [Mason and Carey 1983], and that it should actively involve both the developer *and* the customer [Christensen and Kreplin 1984]. For it to be efficient, it requires close cooperation, and can be successful only when it is based on an actual working system [Somogyi 1981]. Although customers are not very good at stating what they want from a future software system, they are very proficient at criticizing a current system!

A number of techniques have emerged in recent years that are based on this idea. They are classed under the generic term **rapid prototyping** [Smith 1982, Zelkowitz 1984]. The use of these techniques represents a major change in the way software is produced. They rely on an idea borrowed from other engineering disciplines – that of producing a cheap and simplified prototype version of a system rapidly and early in a project. This prototype becomes a learning device to be used by both the customer and the developer and provides essential feedback during the construction of a system specification. The prototyping approach, when compared to current methods, is so dynamic that the difference can be compared to that between interactive and batch oriented systems [Naumann and Jenkins 1982].

Like software testing [Meyer 1978], the main philosophical issue in prototyping is admission of failure; that we, as human beings, no matter how careful in our development practices, are likely to make mistakes. Bally *et al.* [1977] put the idea appropriately in the following words.

> 'In one sense the prototype strategy is an admission of failure, an admission that there will be circumstances in which, however good our techniques and tools for investigation, analysis and design, we shall not develop the right system at the first attempt. But surely this is only realism based on hard experience, theoretically ideal solutions are often far from satisfactory in a very imperfect world.'

One of the objectives of the prototyping approach is to reduce the maintenance effort. There is now considerable evidence [Swanson 1976, Zelkowitz *et al.* 1979, Lientz and Swanson 1980, Lientz 1983] that software maintenance can occupy between 50 to 90% of total project cost during the lifetime of a system. There is increasing empirical evidence [Boehm *et al.* 1984] that prototyping can indeed produce more maintainable products.

Overall, the limited results and experience which have been obtained have been very encouraging. For example, in a reported prototyping experiment [Boehm *et al.* 1984], systems were developed at 40% less cost and 45% less effort than conventional methods. Other researchers have reported even more impressive figures. Scott [1978] has described a system which was estimated to cost $350,000 to develop but was accomplished by a prototype that cost less than $35,000. The figures that have been reported have also supported the contention that prototyping shortens the overall development cycle for software [Berrisford and Wetherbe 1979, Mason and Carey 1983, Bonet and Kung 1984].

1.4 Summary

This chapter has only been introductory in nature. It has pointed out that there are a number of problems associated with conventional software development. Typical problems include the inability to cope with user requirements, and the late visibility of the software product. Software prototyping was presented as one solution to some of these very large problems and some empirical data was presented to support this.