South African
Computer
Journal
Number 20
December 1997

Suid-Afrikaanse
Rekenaar-
tydskrif
Nommer 20
Desember 1997

Computer Science and Information Systems

Rekenaarwetenskap en Inligtingstelsels

The South African Computer Journal
An official publication of the Computer Society of South Africa and the South African Institute of Computer Scientists

Die Suid-Afrikaanse Rekenaartydskrif
'n Amptelike publikasie van die Rekenaarverenging van Suid-Afrika en die Suid-Afrikaanse Instituut vir Rekenaarwetenskaplikes

Editor
Professor Derrick G Kourie
Department of Computer Science
University of Pretoria
Hatfield 0083
dkourie@dos-lan.cs.up.ac.za

Subeditor: Information Systems
Prof Lucas Introna
Department of Informatics
University of Pretoria
Hatfield 0083
lintrona@econ.up.ac.za

Production Editor
Dr Riël Smit
Mosaic Software (Pty) Ltd
P.O.Box 23906
Claremont 7735
gds@mosaic.co.za

World-Wide Web: http://www.mosaic.co.za/sacj/

Subscriptions

	Annual	Single copy
Southern Africa:	R50,00	R25,00
Elsewhere:	$30,00	$15,00

An additional $15 per year is charged for airmail outside Southern Africa

to be sent to:
Computer Society of South Africa
Box 1714 Halfway House 1685

Phone: +27 (11) 315-1319 Fax: +27 (11) 315-2276

Editorial Notes

For two reasons, this edition of SACJ is far later than it ought to have been. The first reason is that there have been some personnel changes in the editorial team. Lucas Introna, having continued for some time as IS editor after transferring to London, asked to be relieved of his duties. Niek du Plooy has kindly agreed to fulfill this role in a temporary capacity until a suitable replacement for Lucas can be found. Due to work pressure, Riël Smit has also withdrawn as production editor, and has been replaced by John Botha. SACJ owes the two retired members a huge debt of gratitude. During his period of tenure, Lucas did sterling work in setting and maintaining a solid standard for IS contributions. Riël put SACJ on a LaTeX path, and has laboured diligently to produce an aesthetically pleasing product. Thanks are also due to Niek and John for their willingness to take over in their respective roles. Until further notice, IS contributors may forward their submissions directly to Niek at his address given on the front inside cover. I shall put successful authors in touch with John for further instructions regarding final preparation of their manuscripts.

The second reason for a delay in this edition has to do with authors who have not scrupulously followed guidelines for producing their final submissions. There have been a variety of problems ranging from missing citations and inappropriate production of figures to incompatible electronic file submissions. All of this, coupled with our new production editor (who—despite an extremely busy schedule—has valiantly climbed a steep LaTeX learning curve) has resulted in an edition that should have been out to press several weeks earlier.

The editorial team will be giving attention to the general matter of format and submission procedures in future. SACJ's citation and reference methods are somewhat archaic and will probably be revised. All the necessary information will be provided on the new SACJ web site at `www.cs.up.ac.za/sacj/`. The site will also contain abstracts of articles in this and future editions.

These are times of conflicting stresses on both the academic and industrial IT communities. They are being felt somewhat more acutely in Southern Africa (and presumably in other developing countries) than in the developed world. Internationally there is tendency to cut back on state financing of universities and a seemingly insatiable demand for IT graduates. Many companies snap up new graduates at attractive salaries, positively discouraging full-time postgraduate studies. International recruitment agencies scour the South African scene for qualified candidates, luring some of our most promising young professionals out of the country. Job-hopping, a drift from academia to industry and from local industry to USA or European industry seems to be the order of the day. Despite the availability of private colleges and institutes, virtual or otherwise, there is a rush of students to university and technikon IT departments, all hoping to get at the IT honey-pot. University administrations are struggling to correct the structural deficiencies of the past and to provide IT departments with sufficient resources to cope with demand. As editor of SACJ, I have no particular competence authoritatively to sum up or analyze these tendencies, but it does seem to me desirable that someone ought to do so. Bodies such as SAICSIT, the CSSA, university authorities, IT industry and state representatives ought actively to pursue joint strategies to ensure that our IT departments are properly resourced and that (non-Zuma) measures are taken to retain graduates in the country. It seems almost redundant to attempt to spell out the consequences of inactivity.

Derrick Kourie
EDITOR

Applying Software Engineering Methods to Instructional Systems Development

Paula Kotzé Ruth de Villiers *
*Department of Computer Science and Information Systems, University of South Africa, PO Box 392, Pretoria, 0001.
E-mail: {kotzep,dvillmr}@alpha.unisa.ac.za

Abstract

The research reported in this paper aims to integrate software engineering approaches with instructional factors in the requirements analysis, design and production phases of instructional software development. The integration has resulted in the evolution of a branch of software engineering called courseware engineering. The paper reports on the results of an independent study we have undertaken into the aspects of software engineering that are appropriate for the development of instructional software systems. Examples of other research efforts combining the disciplines of software engineering and instructional system development are given, where applicable. Two software engineering methods were identified as being particularly useful and appropriate to instructional systems development: prototyping and object-oriented design. Factors that support the use and applicability of these two approaches are discussed and illustrated by a prototype development called FRAMES.

Keywords: *Software engineering, computer-aided instruction, instructional systems development, courseware engineering.*
Computing Review Categories: *K.3.1, K.6.3, D.2.10, D.2.m*

Received: 26/11/1996, Accepted: 12/2/1997, Final version: 12/2/1997.

1 Introduction

Software engineering (SE) is concerned with the development of software systems using sound engineering principles including both technical and non-technical aspects. Over and above the use of specification, design and implementation techniques, human factors and software management should also be addressed. Well-engineered software provides the services required by its users. Such software should be produced in a cost-effective way and should be appropriately functional, maintainable, reliable, efficient and provide a relevant user interface [31, 32].

Computer-aided instruction (CAI) is concerned with the way in which computers can be used to support students engaged in particular educational activities and incorporates a variety of computer-aided instruction and learning modes, for example, formal courseware such as tutorials and drills, and more open-ended software such as simulations, concept maps and microworlds. Over the last decade such systems have proliferated, evolving from simple beginnings, merely in the educational territory, to the realm of complex software systems. As such, they should be designed and developed by applying the established principles of software engineering to instructional systems development (ISD).

The general problem of ISD is therefore to develop appropriate methods for the specification, design and implementation of CAI software systems. The research reported in this paper aims to integrate software engineering approaches with instructional factors in the requirements analysis, design and production phases of instructional software development. The integration has resulted in the evolution of a branch of software engineering called courseware engineering [3, 30]. Two SE methods are identified as being particularly useful and appropriate to ISD: prototyping and object-oriented design. Factors that support the use and applicability of these two approaches are discussed. These approaches were implemented in the development of the FRAMES prototype.

2 Software Engineering Methods

One of the cornerstones of SE is the software life cycle, describing the activity stages that take place from the initial concept formation for a software system, up to its implementation and eventual phasing out and replacement. Complementing these life cycle models are a number of design and development models.

Life cycle models

A multitude of general software engineering life cycle (or process) models exist, some of which are variations of others. Two examples are the so-called 'waterfall model' and the 'spiral model'.

- **Waterfall model**: In the waterfall approach [9, 12, 32] the software process is viewed as being made up of a number of stages or activities such as requirements specification, software design, software implementation, testing, operation and maintenance, etc. Each activity serves as input to the next. The waterfall model requires that rigorous requirements analysis be done before design, and rigorous design before implementation. A disadvantage is thus that it suits an approach to design in which all requirements for a system have to be known before system development is begun.

  Real projects, however, rarely follow the sequential flow that this model proposes. The strict sequential sequence of the waterfall and other similar models does not lend itself to projects that are characterized by un-

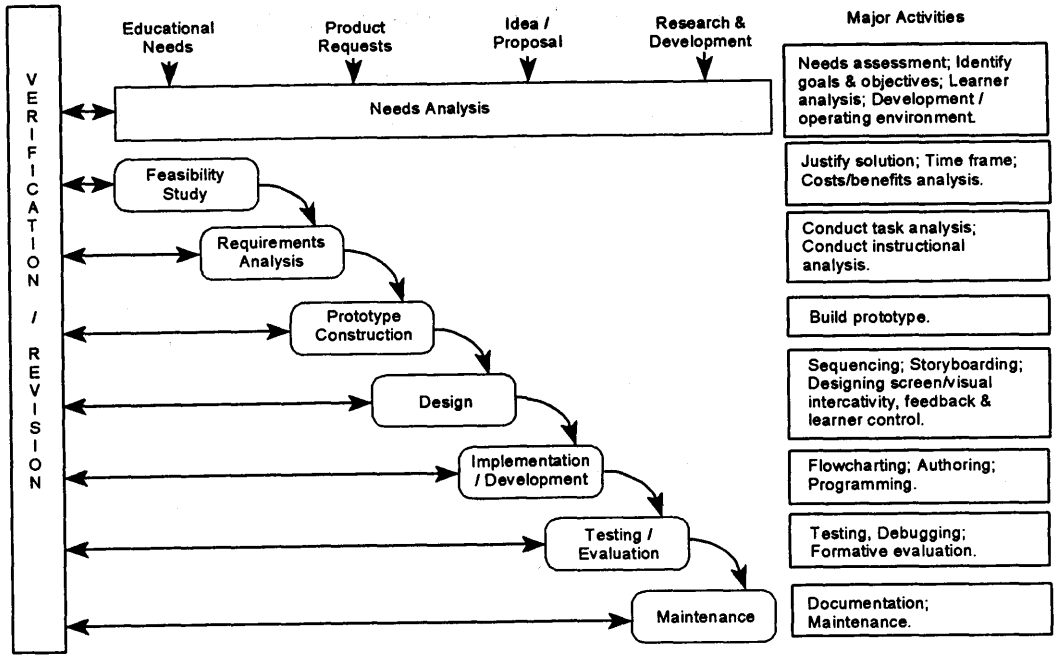

Figure 1. Chen and Shen's life cycle model

[10, p 11]

certainty and requirements that are ill-defined in the early stages of development. It is too rigid for use in situations of changing or incomplete requirements. This also applies to CAI systems development.

The behaviour of a CAI system is highly dependent on the domain knowledge modelled within the system. The tasks a student-user will perform, or that a teacher wishes a student to perform, are often not known until the student and/or teacher is familiar with the system on which the tasks are performed. A second drawback of this process model is that it does not promote the use of notations and techniques which support the user's perspective of the CAI system—it is very difficult for even an expert on human cognition to predict the cognitive demands that an abstract design would require of the student-user if the notation for the design does not portray the kind of information the student must recall in order to interact with the system [16].

- **Spiral model**: An alternative to the waterfall approach is Boehm's spiral model [7] which is essentially an iterative model. Its key characteristic is an assessment of management risk items at various stages of the project and the initiation of actions to counteract these risks. Before each cycle, a review procedure judges whether to move on to the next cycle in the spiral. A cycle of the spiral commences by elaborating objectives such as performance, functionality, and so on. Alternative ways of achieving these objectives and constraints are then enumerated, followed by an assessment of each objective. This typically results in the identification of sources of project risk. The next step is to evaluate these risks by activities such as more detailed analysis, prototyping, simulation, etc. After risk-evaluation a development model (or a combination of models) for the system is chosen.

  A major deficiency of the spiral approach is that it requires an approach to system development that is both incremental and interactive—the cycles must be seen to be achieving project aims. Managing a medium-to-large sized spiral-driven project, especially when considering scheduling and product delivery aspects, is very difficult. The development time required for typical interactive ISD can get totally out of hand following such an approach.

Integrating SE aspects with androgogic activities related specifically to the design and development of instructional systems resulted in a number of specialised life cycle models specifically aimed at the development of computer-based instructional systems. Two examples are:

- **Chen and Shen's model**: Chen and Shen [10] propose a life cycle model similar to the waterfall model but aimed specifically at the development of CAI with the joint objectives of producing high quality products and development effectiveness. Verification and revision occur after each phase, resulting in an iterative, cyclic process, as illustrated in Figure 1.
- **Tennyson's ISD[4]**: Tennyson's Fourth Generation Instructional Systems Design Model (ISD[4]) [33], as illustrated in Figure 2, advocates the employment of advances in cognitive science and intelligent programming techniques to automate instructional systems development. It focuses on explicit production rules or neural network methods in terms of development activities rather than on the sequencing of phases. In-

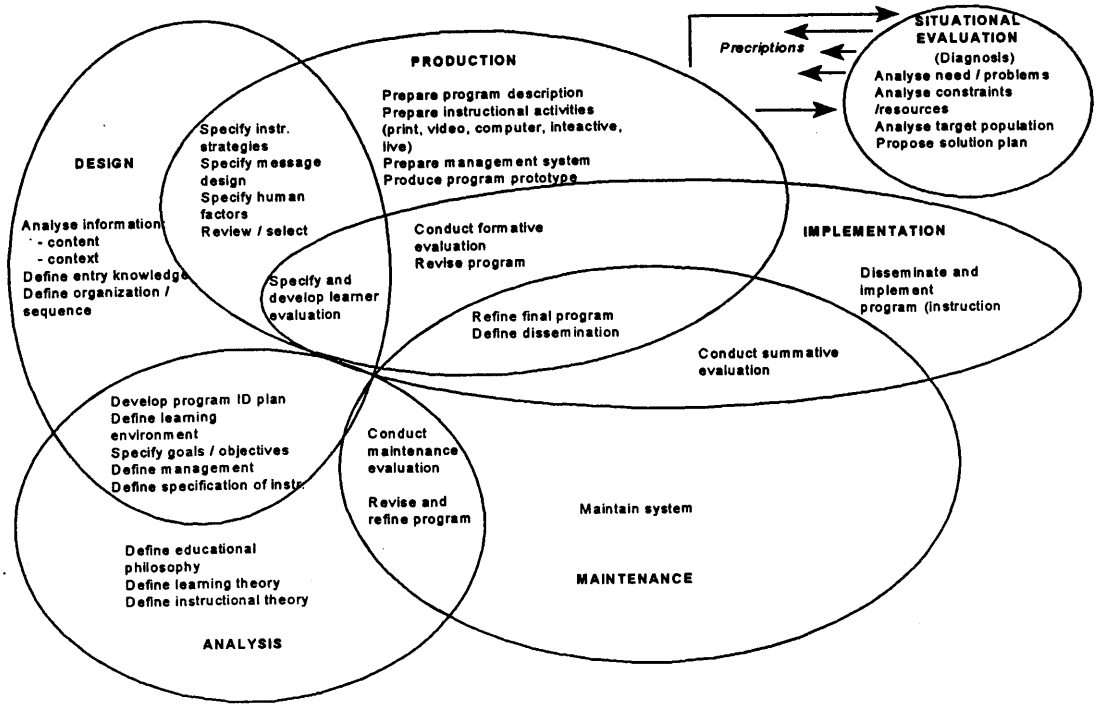

Figure 2. ISD[4]

[33, p 4]

stead of a system consisting of a confederation of abstract, often ill-defined, or expert controlled procedures, ISD[4] proposes systems of explicit rules controlled by contextual or situational problem solving evaluations. ISD[4] consists of two main components—a situational evaluation component that assesses the situation and then offers an expert prescription, and a knowledge base that includes five domains of interactive instructional development elements. These elements are by nature overlapping in their respective activities, as illustrated in Figure 2. Thus, rather than a step-by-step instructional development process as advocated by the waterfall approach or the model of Chen and Shen, ISD[4] prescribes the authoring activities to fit the given situation.

Development models

Detailed software process models are still the subject of research, but a number of general models or paradigms of software development can be identified as supporting these process models [16, 21, 32]. Two of these approaches have been widely used:

- **Exploratory programming** (also known as **evolutionary prototyping**): A working system is developed as quickly as possible and then modified until it performs in an adequate way. This approach is used to a great extent in artificial intelligence systems development where a detailed requirements specification cannot be formulated, and where adequacy rather than correctness is the aim of the systems designers. The disadvantage of this approach relates to the encapsulation of design decisions. Firstly, some of the earlier decisions may have been wrong and may never be removed from the system. Secondly, because the behaviour of an interactive system is highly dependent on the knowledge modelled within the system, earlier versions of the system will not include all of the knowledge to be included in the completed system.
- **Throw-away prototyping**: The software process starts off in a similar way to exploratory programming in that the first phase of development involves the development of a program for user experiment. The objective of the development is, however, to establish system requirements. Rapid application approaches are a case in point. Screen layouts resembling those of a complete system may quickly be developed and shown to the users for comment. The prototyping process is followed by re-design in order to implement the full software system. Suffering from the same drawbacks as exploratory programming, it has a further disadvantage in that it may concentrate only on the surface features of the design, rather than on deeper issues and the functioning of the interface, and does not, on its own, guarantee that the software produced exhibits the required interaction qualities.

To compensate for the limitations of the prototyping approaches, and to support the waterfall and spiral models in the requirements and design activity phases, a number of researchers in recent years have advocated the use of two more approaches in the design process of interactive systems: formal transformation and system assembly from reusable components.

- **Formal transformation**: A formal or abstract specification of the software system is developed and

then transformed, by means of correctness-preserving transformations, to an implemented software system. The principal value of using formal specification techniques in the software development process is that it compels an analysis of the system requirements at an early stage. Correcting errors at this stage of development is much cheaper than modifying a delivered system. A range of abstract modelling approaches for interactive systems are reported in [1, 13, 15, 18, 21, 27, 28, 32].

The formality of these models is intended to assure the exploration of the consequences of the design without constructing prototypes or other working models of the design, as well as algorithmic manipulation of the design.

This development technique is, however, still in its infancy and mainly used in high-budget safety-critical system developments, where correctness-proving is a major criterion.

- **System assembly from reusable components**: The system development process is either a total reuse process using components which already exist in assembling the new system, or it involves the reuse of available components applicable to the envisaged system while additional components are developed using any other development approach.

  A trait of an engineering discipline is that it is founded upon an approach to system design which makes maximum use of existing components. Design engineers base their designs on components which are common to, and tried and tested in other systems of the same, or similar, nature. A number of reuse approaches are reported in [4, 20, 21, 34].

  Development of reusable components takes much more effort and specific design than for once-off systems development. Issues still being actively researched include frameworks for reusable components [21] and other architectural issues such as cross-platform design-compliant assembly of components.

Notwithstanding the disadvantages of the various prototyping approaches, they are still the most viable software development models for the development of interactive instructional systems. There are several reasons for this, the first being the high interactivity that characterizes instructional software. It requires that the user interface and response mechanisms be experienced, expanded and refined hands-on, rather than merely from paper-based storyboards. Another reason relates to the computer literacy levels of the intended end-users, i.e. the learners, who frequently are computer novices. Unless they are at ease with the human-computer interface, the software will do little to achieve its instructional and learning ends. Prototypes can be pilot-tested by such end-users. A third factor is that the clients (instructors, teachers, trainers, etc.) requiring instructional software for their target group of learners, are often inexperienced computer users with no background in formal specification methods. The use of formal transformation techniques would therefore be totally inappropriate.

Prototyping allows for early evaluation by instructors, trainers, teachers, subject-matter experts, peers, etc. Visual perception and hands-on experience of part of an operational system often results in the instructor-client modifying the objectives and strategies.

The major purposes of conventional software are data processing and information processing, where defined activities occur in a predefined sequence. Instructional software, by contrast, comprises synthesis, presentation, practice and assistance facilities in the complex realm of human cognition, and has a high level of human-computer interactivity. Whether in a situation of program-control where the flow is branched deterministically according to user-response, or in a situation of user-control where the learner may branch or browse at will, the sequence of events and activities varies greatly. CAI prototypes can play a vital role in demonstrating proposals on-screen, thus clarifying actual requirements and identifying misconceptions and potential problems at an early stage. Not only should basic aspects such as screen layout and colours be scrutinised, but also the strategies for control and navigation through the material. Usability factors, such as learnability and consistency can be evaluated, also interface aspects such as coherence of textual and visual displays, and accessibility of facilities.

Particularly when an instructional software package is innovative, prototyping is required at the design and programming stages in order to ensure feasibility of intentions, to refine requirements, to reduce excessive written descriptions, to determine the optimal navigation and control strategies hands-on, and also to ensure that an appropriate programming approach is used for implementation [14].

Development tools or authoring environments should offer modularity, thus facilitating the removal, addition or adaptation of a segment without affecting other segments or the unit as a whole, and plasticity, the ability to make changes easily. If exorbitant time and costs are incurred in developing a prototype, the process is not cost-effective. An ISD prototype may either be evolutionary, i.e. a limited version of the final product later developed through to full functionality, or else a throwaway. In the latter case, the software used to build the prototype may not be the same as that used for the final system.

It is also interesting to note that prototype production forms a central part of Chen and Shen's life cycle model, and is also listed as one of the activities in the production phase of ISD[4]. Other researchers advocating the use of prototyping approaches for ISD include Black and Hinton [5, 6], Gray and Black [17], Lantz [23], Tripp and Bichelmeyer [35], and Wong [36].

Furthermore, although envisaged by many, there is still no common base of reusable components of CAI software which is widely documented and which can be used when developing a CAI system with similar functionality [21].

Design models

Software design is in essence a problem-solving task. It is more important to design a solution that will achieve

its purpose in doing the required job effectively, than to achieve elegance and efficiency at the expense of accuracy and reliability. A designer needs to abstract the critical features of a system, so as to concentrate initially on building a logical model of the system rather than becoming over-involved with detailed design and physical implementation at an early stage.

Various methodologies and representations are available to facilitate the processes of analysis, design, and system modelling, in particular, the process-oriented, data-oriented, and object-oriented approaches:

- **Process-oriented**: The process-oriented approach centres around the events, procedures and flows that comprise a traditional procedural software system. Such applications are characterised by conventional data flow and updating of data stores. Events trigger processes, and processes call other processes, sequentially or selectively. The process-oriented approach is epitomised by concepts and tools such as top-down design, functional decomposition, transaction analysis, data-flow diagrams, structure charts and input-output transformations. It tends to discount evolutionary changes.
- **Data-oriented**: The data-oriented approaches are based on the philosophy that data is more stable and unchanging than processes. The underlying principles are enterprise analysis and relational database theory, and key concepts are entities, attributes, relationships and normalisation.
- **Object-oriented**: The latest advancement is the object-oriented approach [2, 8, 9, 11, 12, 29], which integrates aspects of, and uses formalisms from, both the other major methodologies, and uses certain concepts from object-oriented programming languages. It is based on objects, which encapsulate both data and operations (processes) on that data. An object is a real-world entity whose processes and attributes are modelled in a computerised application. In object-oriented programming languages, computation is achieved when messages are passed to the objects in the program, and a central aspect is the abstract data type (ADT), which permits operations to be performed on an object without being implementation-specific. Objects incorporating data are identified as data entities and not as specific data structures.

Conventional data-flow and process-linkage are not characteristic features of instructional and learning software and such software therefore does not lend itself to the process-oriented approach. CAI courseware and environments comprise relatively few objects and components when compared to the large, complex systems developed by the object-oriented methodology. Nevertheless, the object-oriented strategies outlined can be beneficially applied in the analysis, design and development of instructional and learning software.

Budgen [9] describes an object as an entity which possesses a state, exhibits behaviour, and has a distinct identity. Sommerville [32, p 194] proposes the following definition: "An object is an entity which has a state (whose representation is hidden) and a defined set of operations which operate on that state. The state is represented as a set of object attributes".

Analysis of classical CAI tutorials, simulations, drill-and-practice software, and state-of-the-art user-controlled interactive learning environments reveals distinct design objects, or components, which possess unique identities, certain attributes and relationships, and have operations performed on them, i.e. much CAI software explicitly, or implicitly, consists of instructional components [25]. Many CAI systems are comprised mainly of instructional presentations and exercise/question segments. The main processing activities are determination of which unit / segment / example / exercise to present or do next, and the assessment of student responses. The means of determination depends on the locus of end-user control whether program-control, learner-control, or a combination thereof. The various and varied instructional activities and learning experiences comprising learning segments, example presentations, practice exercises and assessment activities whether in textual or graphic form, whether requiring active learner participation or passive perusal, can readily be perceived as objects. The objects are separate, yet strongly interrelated and it is appropriate to implement them in an object-oriented design. Such component-based instructional systems can best be implemented by an object-oriented design [14]. The object-oriented approach thus appears to be the most appropriate software engineering development methodology for ISD.

Even in the traditional life cycle models such as the waterfall model, the distinction between the system design (broad design) and the program design (detailed design/coding) can become blurred. In the object-oriented approach, however, the boundary is even more indistinct, because both top-down analysis and bottom-up program development occur simultaneously or, at least, iteratively. The three traditional activities of analysis, design, and implementation are all present, but the joints between are seamless. The unifying factor is the prime role played by objects and their interrelationships. Modelling is prominent in object-oriented design, the basic architecture being assembled from models of the entities and the relationships between them. Reuse is a feature of object-oriented design, since the prominent class and inheritance features lend themselves to code reuse.

CAI software incorporates well-defined objects, both concrete and abstract, and particularly in situations with an initial lack of precise specifications, the procurement of instructional software can be expedited and facilitated by a development process incorporating evolutionary prototyping. This requires a life cycle model emphasizing overlap and evolution with explicit incorporation of a prototyping phase. Chen and Shen's model, ISD^4, as well as other similar life cycle models, for example the Wong prototyping model [36], the Booch model [8] and the Henderson-Sellers and Edward's fountain model [19], all incorporate these requirements.

3 An Application of Software Engineering Methods to CAI

The principles outlined in section 2, in particular, prototyping and an object-oriented approach, were applied in the design and development of FRAMES, an interactive practice environment in Theoretical Computer Science. A complete description of the development of FRAMES and examples of its components are given in [14]. The target group is 1st-level BSc students and the context is relations, a topic in discrete mathematics. A prototype implementation was developed according to an evolutionary prototyping life cycle and in line with the object-oriented methodology.

The requirements analysis phase in instructional systems development is broad, incorporating decisions regarding the general characteristics, both instructional- and computing-oriented, of the required software. The goal in developing FRAMES was to produce a practice-environment, a kind of androgogic activity box, providing a variety of useful instructional and learning activities, and to develop it using a software engineering approach. The problem domain, namely relations, was modelled with an entity-relationship-attribute diagram to identify its entities and the relationships between them. The instructional design approach selected for FRAMES is based on a cognitive theory, Merrill's Component Display Theory (CDT), whereby all instructional and learning activities (termed instructional transactions or components) [24, 26] are categorized on a 2-dimensional grid according to the content taught and the type of performance expected from learners. Thus FRAMES consists of components which are well-modelled by objects according to the object-oriented approach. These components comprise both presentation of tutorial matter and practice opportunities.

Correct application of CDT results in instructional products which learners can actively control to meet their own needs, according to learning style or stage of study. This active role of the learner was the prime factor in determining the design and screen layout of FRAMES. The idea was not to produce a formal tutorial, but to develop a structure comprising a variety of components available as exercises for the learner, where each instructional component may be perceived as an object. Each learner makes his/her own selection of content, strategy, quantity and sequence, thus constructing screens comprising his/her own chosen set of components. This design lends itself to implementation via an object-oriented development methodology, in which each instructional transaction / component comprises an object. The objects are closely interrelated, since:

- the more complex, integrated problems are comprised of simpler objects;
- certain objects have the same format, because they incorporate the same kind of analysis operation applied to different examples from the domain;
- other objects apply different kinds of analysis operations, but to the same example data;
- certain objects cover exactly the same material but in different instructional modes, thus facilitating reinforcement.

Although the high degree of user control permits each learner to create his/her own diverse combination of activities, the control structure and nature of activities lend themselves to reusability of structures and formats. FRAMES has a high reuse factor of its components.

Most transactions are interactive objects, requiring response on the part of the learner; in these cases meaningful feedback is given in the form of judgement and assessment. Some of these are open-ended, in that they require the learner to synthesise examples. The environment also contains useful auxiliary objects such as definitions, graphic aids, help overlays and control icons.

An initial throwaway prototype was used to test the feasibility and visual aspects of certain innovative techniques. The major evolutionary prototype was produced by a design-and-refine paradigm which facilitated evaluation of the interaction, clarified requirements and identified potential problems.

The FRAMES prototype is a highly interactive, individualised practice environment covering about 40% of the material intended for inclusion in the production model. FRAMES was programmed in TenCORE 5.0 by programmers of CENSE (Centre for Software Engineering at Unisa) and runs under DOS.

4 Conclusion

This article overviewed general software engineering models, tools and techniques, and investigated their applicability to instructional systems development.

The main difference between conventional and instructional systems development is important—conventional systems primarily involve data processing, while instructional systems attempt to stimulate human cognition by means of interactive presentation, practice opportunities, and support. Conventional systems must support tasks such as data capturing and report generation through task-appropriate user interfaces, while instructional systems should promote learning through a good user interface, proper knowledge modelling and navigation through such recorded knowledge.

The absence of data-flow interaction in instructional systems calls for a development process based more on refinement of presentation and knowledge navigation structures, than on formal data flow and algorithmic models. A life cycle model which includes evolutionary prototyping, such as the approach followed with FRAMES, therefore appears to be most appropriate for the development of instructional software, so that initially fuzzy requirements can be refined and the initial working version can be modified and expanded towards a final operational CAI product.

The object-oriented methodology proves itself to be, in the terms of Korson & McGregor [22], "a unifying paradigm", which is appropriate for the analysis and representation of CAI. CAI systems consist of compo-

nents which can be well-modelled by objects in an object-oriented approach. Viewing a system as object-based provides a more versatile foundation than a view based fundamentally on data modelling or on its functions and procedures. Although user-input plays a major role in determining the path through instructional software, there is little conventional data flow. The concept of an object is therefore a utilitarian approach, which brings together such varied items as concrete objects, abstract objects, data, processes, and environmental entities external to the software (yet vital components of the system), such as the human user. Incorporation of the user as an object is particularly beneficial in CAI, due to its highly interactive and individualised nature. Tools and representations of the object-oriented approach were used in the analysis, design and documentation of the instructional software incorporated in the FRAMES system.

Although experience has shown that re-design and re-implementation are almost always inevitable in order to produce a reusable framework, it is hoped that object-based control structures developed for specific applications, as demonstrated in FRAMES, can eventually be used as generic, content-free shells to present formal instruction or practice exercises in different instructional modes in varying subjects and courses. This would capitalise on the modularity and reuse potential inherent in an object-oriented design.

References

1. G D Abowd. *Formal Aspects of Human-Computer Interaction.* DPhil Thesis, Oxford University, Programming Research Group, 1991.
2. D Bell, I Morrey and J Pugh. *Software Engineering: A Programming Approach.* Prentice Hall International, Hemel Hempstead, 1992.
3. M N Bessagnet, T Nodedot, G Gouarderes and JJ Rigal. A new approach: courseware engineering. In: *Computers in Education*, edited by A McDougall and C Dowling. Elsevier Science Publishers, Amsterdam, 1990.
4. T J Biggerstaff and A J Perlis (Eds). *Software Reusability, Volumes I & 2.* Addison-Wesley, Reading MA, 1989.
5. T R Black. Prototyping CAL courseware: a role for computer-shy subject experts. In *Aspects of Educational Technology Vol XXI, Designing New Systems and Technologies for Learning*, edited by H Mathias, H Rushby and R Budgett. Kogan Page, London, 1988.
6. T R Black and T Hinton. Courseware design methodology: the message from software engineering. In *Aspects of Educational Technology Vol XXII, Promoting Learning*, edited by C Bell, J Davies and R Winders. Kogan Page, London, 1989.
7. B W Boehm. A spiral model of software development and enhancement. *IEEE Computer*, **21**(5), 61 – 72, (1988).
8. G Booch. *Object-Oriented Analysis and Design: with Applications.* Benjamin/Cummings Publishing Company, Redwood City, 1994.
9. D Budgen. Software Design. Addison-Wesley, Wokingham, 1994.
10. J W Chen and C Shen. Software engineering: a new component for instructional software development. *Educational Technology*, **29**(9), 9 – 15, (1989).
11. P Coad, D North and M Mayfield. *Object Models: Strategies, Patterns and Applications.* Prentice Hall, Englewood Cliffs, 1995.
12. S A Conger. *The New Software Engineering.* Wadsworth Publishing Company, Belmont, 1994.
13. A M Dearden. *The use of Formal Models in the Design of Interactive Case Memory Systems.* DPhil Thesis, Human-Computer Interaction Group, University of York (UK), 1995.
14. M R De Villiers. *Integrating a Software Engineering Approach in Instructional Software Development—Illustrated by a Prototype in Theoretical Computer Science.* MSc Dissertation, Department of Computer Science, University of South Africa, 1995.
15. A Dix. *Formal Methods for Interactive Systems.* Academic Press, London, 1991.
16. A Dix, J Finlay, G Abowd and R Beale. Human-Computer Interaction. Prentice-Hall, Hemel Hempstead, 1993.
17. D E Gray and T R Black. Prototyping of computer-based training materials. *Computers in Education*, **22**(3), 251 – 256, (1994).
18. M D Harrison and H Thimbleby (Eds). *Formal Methods in Human-Computer Interaction.* Cambridge University Press, Cambridge, 1990.
19. B Henderson-Sellers and J M Edwards. The object-oriented systems life cycle. *Communications of the ACM*, **33**(9), 142 – 159, (1990).
20. R Johnson and B Foote. Designing reusable classes. *Object-Oriented Programming*, **1**(2), 22 – 35, (1988).
21. P Kotzé. *The Use of Formal Models in the Design of Interactive Authoring Support Environments.* DPhil Thesis, Human-Computer Interaction Group, University of York(UK), 1997.
22. T Korson and J D McGregor. Understanding object-oriented: a unifying paradigm. *Communications of the ACM*, **33**(9), 40 – 60, (1990).
23. K E Lantz. *The Prototyping Methodology.* Prentice-Hall, Englewood Cliffs, (no date).
24. M D Merrill. Component Display Theory. In *Instructional Design Theories and Models: An Overview of their Current Status*, edited by C M Reigeluth, Lawrence Erlbaum Associates, Hillsdale, 1983.
25. M D Merrill. Applying component display theory to the design of courseware. In *Instructional Designs for Microcomputer Courseware*, edited by D H Jonassen. Lawrence Erlbaum Associates, Hillsdale, 1988.
26. M D Merrill. Instructional Design Theory for Automated Instructional Development. In *Emerging Computer Technologies in Education: Selected papers from the International Conference on Computers in Education (ICCE) Taiwan, 1993*, edited by T Chan and A Self. AACE, Charlottesville, (1993).
27. P Palanque and R Bastide (Eds). *Proceedings of the Eurographics Workshop in Toulouse France, June 1995.* Springer-Verlag, Wien, 1995.
28. F Paternó (Ed). *Interactive Systems: Design, Specification and Verification.* Springer-Verlag, Berlin, 1995.
29. S R Schach. *Classical and Object-Oriented Software Engineering.* Aksen Associates Inc. Publishers, Boston, 1996.
30. J Schoenmaker, E Nienhaus, J Scholten and J Titulaer. A methodology for educational software engineering. In *Computers in Education*, edited by A McDougall and C Dowling. Elsevier Science Publishers, Amsterdam, 1990.

31. B Shneiderman. *Designing the User Interface.* Addison-Wesley, Reading, 1992.

32. I Sommerville. *Software Engineering.* Addison-Wesley Publishing Company, Wokingham, 1996.

33. R D Tennyson. Knowledge base for automated instructional system development. In *Automating Instructional Design, Development, and Delivery*, edited by R D Tennyson. Springer Verlag, Berlin, 1994.

34. W Tracz (ed). *Software Reuse: Emerging Technology.* IEEE Computer Society Press, Washington, 1988.

35. S D Tripp and B Bichelmeyer. Rapid prototyping: an alternative instructional design strategy. *Educational Technology, Research and Development*, **38**(1), 31 – 44, (1990).

36. S C Wong. Quick prototyping of educational software: an object-oriented approach. *Journal of Educational Technology Systems*, **22**(2), 155 – 172, (1993).

Notes for Contributors

The prime purpose of the journal is to publish original research papers in the fields of Computer Science and Information Systems, as well as shorter technical research notes. However, non-refereed review and exploratory articles of interest to the journal's readers will be considered for publication under sections marked as Communications or Viewpoints. While English is the preferred language of the journal, papers in Afrikaans will also be accepted. Typed manuscripts for review should be submitted in triplicate to the editor.

Form of Manuscript

Manuscripts for *review* should be prepared according to the following guidelines.

- Use wide margins and $1\frac{1}{2}$ or double spacing.
- The first page should include:
  - title (as brief as possible);
  - author's initials and surname;
  - author's affiliation and address;
  - an abstract of less than 200 words;
  - an appropriate keyword list;
  - a list of relevant Computing Review Categories.
- Tables and figures should be numbered and titled.
- References should be listed at the end of the text in alphabetic order of the (first) author's surname, and should be cited in the text in square brackets [1–3]. References should take the form shown at the end of these notes.

Manuscripts accepted for publication should comply with the above guidelines (except for the spacing requirements), and may be provided in one of the following formats (listed in order of preference):

1. As (a) LaTeX file(s), either on a diskette, or via e-mail/ftp – a LaTeX style file is available from the production editor;
2. As an ASCII file accompanied by a hard-copy showing formatting intentions:
   - Tables and figures should be original line drawings/printouts, (not photocopies) on separate sheets of paper, clearly numbered on the back and ready for cutting and pasting. Figure titles should appear in the text where the figures are to be placed.
   - Mathematical and other symbols may be either handwritten or typed. Greek letters and unusual symbols should be identified in the margin, if they are not clear in the text.

   Contact the production editor for markup instructions.
3. In exceptional cases camera-ready format may be accepted – a detailed page specification is available from the production editor;

Authors of accepted papers will be required to sign a copyright transfer form.

Charges

Charges per final page will be levied on papers accepted for publication. They will be scaled to reflect typesetting, reproduction and other costs. Currently, the minimum rate is R30-00 per final page for LaTeX or camera-ready contributions that require no further attention. The maximum is R120-00 per page (charges include VAT).

These charges may be waived upon request of the author and at the discretion of the editor.

Proofs

Proofs of accepted papers in category 2 above may be sent to the author to ensure that typesetting is correct, and not for addition of new material or major amendments to the text. Corrected proofs should be returned to the production editor within three days.

Note that, in the case of camera-ready submissions, it is the author's responsibility to ensure that such submissions are error-free. Camera-ready submissions will only be accepted if they are in strict accordance with the detailed guidelines.

Letters and Communications

Letters to the editor are welcomed. They should be signed, and should be limited to less than about 500 words.

Announcements and communications of interest to the readership will be considered for publication in a separate section of the journal. Communications may also reflect minor research contributions. However, such communications will not be refereed and will not be deemed as fully-fledged publications for state subsidy purposes.

Book reviews

Contributions in this regard will be welcomed. Views and opinions expressed in such reviews should, however, be regarded as those of the reviewer alone.

Advertisement

Placement of advertisements at R1000-00 per full page per issue and R500-00 per half page per issue will be considered. These charges exclude specialized production costs which will be borne by the advertiser. Enquiries should be directed to the editor.

References

1. E Ashcroft and Z Manna. 'The translation of 'goto' programs to 'while' programs'. In *Proceedings of IFIP Congress 71*, pp. 250–255, Amsterdam, (1972). North-Holland.
2. C Bohm and G Jacopini. 'Flow diagrams, turing machines and languages with only two formation rules'. *Communications of the ACM*, 9:366–371, (1966).
3. S Ginsburg. *Mathematical theory of context free languages*. McGraw Hill, New York, 1966.

South African Computer Journal

Number 20, December 1997
ISSN 1015-7999

Suid-Afrikaanse Rekenaar-tydskrif

Nommer 20, Desember 1997
ISSN 1015-7999

Contents