SOFTWARE ENGINEERING DEVELOPMENT METHODOLOGIES APPLIED TO COMPUTER-AIDED INSTRUCTION

Paula Kotzé and Ruth de Villiers
Department of Computer Science and Information Systems
University of South Africa, P O Box 392, Pretoria, 0001
Email: {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 (SE) called courseware engineering. The paper reports on the results of an independent study we have undertaken into the aspects of SE that are appropriate for the development of courseware. Examples of other research efforts combining the disciplines of SE and instructional system development (ISD) are given, where applicable. Two SE methods were identified as being particularly useful and appropriate to ISD: prototyping and object-oriented (OO) design. Factors that support the use and applicability of these two approaches are discussed.

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 [Shneiderman 1992, Sommerville 1992].

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 software 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. Two SE methods are identified as being particularly useful and appropriate to ISD: prototyping and object-oriented (OO) design. Factors that support the use and applicability of these two approaches are discussed.

Software Engineering Models

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.

Figure 1: Chen and Shen's life cycle model [Chen 1989, p 11]

Life cycle models

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

- *Waterfall model*: In the waterfall approach [Budgen 1994, Conger 1994, Sommerville 1992] the software process is viewed as 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. One disadvantage of this approach is that it suits a principled approach to design where all requirements for a system have to be known before system development is begun.

  The behaviour of a CAI system is highly dependent on the domain knowledge modelled within the system. The tasks a user will perform are often not known until the user is familiar with the system on which he performs them. 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.

- *Spiral model*: An alternative to the waterfall approach is Boehm's spiral model [Boehm 1988] 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.

Combining SE aspects with those 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:

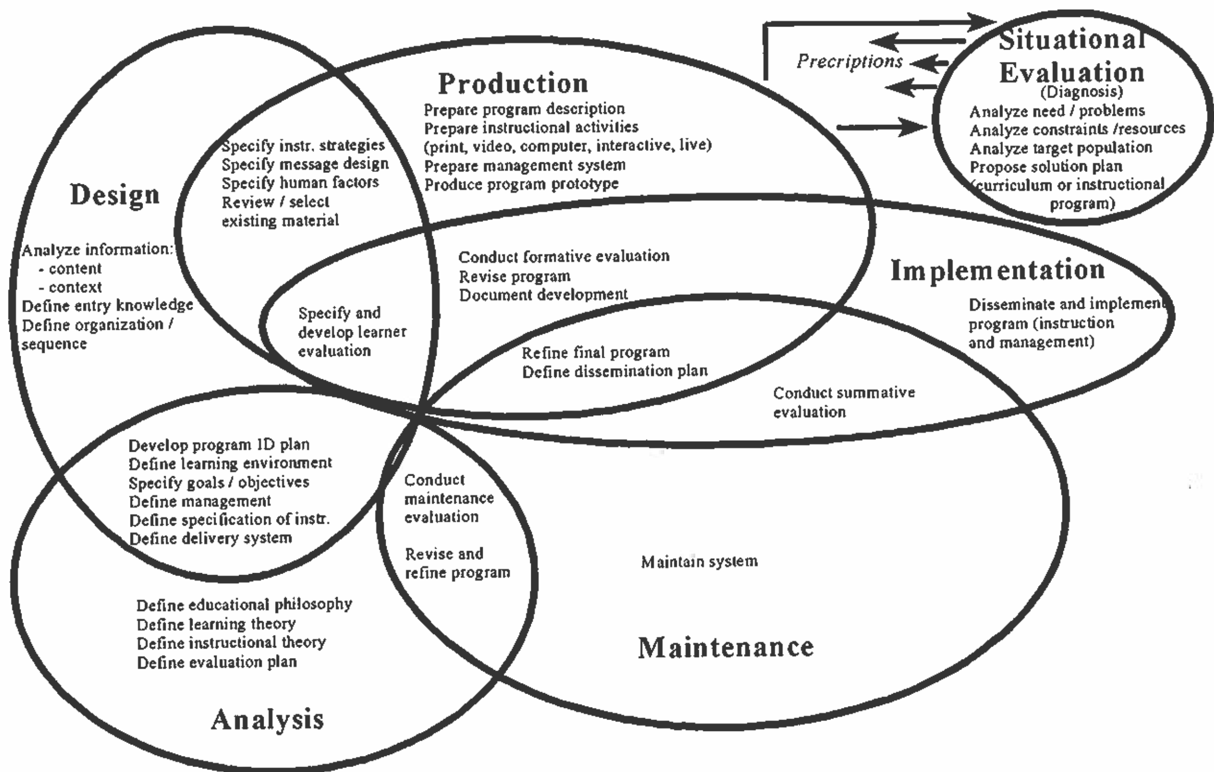

Figure 2: ISD[4] [Tennyson 1994, p 4]

- *Chen and Shen's model*: Chen and Shen [Chen 1989] propose a life cycle model similar to the waterfall model but aimed specifically at the development of CAI with the joint objective 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]) [Tennyson 1994], as illustrated in Figure 2, advocates the employment of advancements from cognitive science and intelligent programming techniques to automate instructional system development. It focuses on explicit rules in terms of development activities rather than on the sequencing of phases.

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 [Dix 1993, Sommerville 1992]. 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. The prototyping process

is followed by re-implementation of the software to produce the desired 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 [Abowd 1990, Dearden 1995, Dix 1991, Harrison 1990, Palanque 1995, Paternó 1995, Sommerville 1992].

  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.

- *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 available components applicable to the envisaged system are reused 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 with, and tried and tested in other systems of the same, or similar, nature. A number of reuse approaches are reported in [Biggerstaff 1989, Johnson 1988, Tracz 1988].

Notwithstanding the disadvantages of the prototyping approaches, they are still the most viable options for the development of interactive CAI systems. The reason for this is firstly determined by the high interactivity that characterises instructional software in relation to the computer literacy levels of the primary developers of such software, as well as that of the end-users of the completed product. The intended end-users are frequently laymen with respect to computer usage and unless they are at ease with the human-computer interface, the software will do little to achieve its instructional and learning ends. Prototypes are instrumental in pilot-testing by such end-users. Prototyping also allow 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. Also, the developers of such system are, more often than not, novice computer users, with absolutely no background in formal specification methods. The use of formal transformation techniques would therefore be totally inappropriate.

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 errors 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 a piece of instructional software breaks new ground, 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.

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.

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. 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 [Black 1988, Black 1989], Gray and Black [Gray 1994], Lantz [Lantz, no date] Tripp and Bichelmeyer [Tripp 1990], and Wong [Wong 1993].

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 properly, 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:

- The *process-oriented paradigm* 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 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.

- The latest emerging methodology is the *object-oriented paradigm* [Bell 1992, Booch 1994, Budgen 1994, Coad 1995, Conger 1994, Schach 1996], 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 do not feature heavily in instructional and learning software and such software therefore does not lend itself to the process-oriented paradigm. Object-oriented development has been used in large, complex systems, compared to which CAI courseware and environments comprise relatively few objects and components. Nevertheless, the strategies outlined can be beneficially applied in the analysis, design and development of instructional and learning software.

Budgen [Budgen 1994] describes an object as an entity which possesses a state, exhibits behaviour, and has a distinct identity. Sommerville [Sommerville 1992, 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 is explicitly, or implicitly, consisting of instructional components [Merrill 1988]. 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. The OO paradigm 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 systems design (broad design) and the program design (detailed design/coding) can become blurred. In the OO paradigm, 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 OO 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, its procurement can be expedited and facilitated by a development process incorporating evolutionary prototyping. Combining the essence of prototyping and the OO paradigm 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 [Wong 1993], the Booch model[Booch 1994] and the Henderson-Sellers and Edward's fountain model [Henderson 1990], all incorporate these requirements.

Conclusion

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

A life cycle model which includes evolutionary prototyping 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 [Korson 1990], "a unifying paradigm", which is appropriate for the analysis and representation of CAI. 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 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. The tools and representations of the OO paradigm can be of great value in the analysis, design and documentation of instructional software.

It is hoped that object-based control structures developed for specific applications 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

[Abowd 1991] Abowd G D. 1991. *Formal Aspects of Human-Computer Interaction.* DPhil. Thesis, Oxford University, Programming Research Group.

[Bell 1992] Bell D, Morrey I & Pugh J. 1992. *Software Engineering: A Programming Approach* (2nd ed.). Hemel Hempstead: Prentice Hall International (UK) Ltd.

[Biggerstaff 1989] Biggerstaff T J & Perlis A J (Eds). *Software Reusability.* Volumes 1 & 2. Reading MA: Addison-Wesley.

[Black 1988] Black T R. 1988. 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 & R Budgett. London: Kogan Page.

[Black 1989] Black T R & Hinton T. 1989. Courseware design methodology: the message from software engineering. In: *Aspects of Educational Technology Vol XXII, Promoting Learning*, edited by C Bell, J Davies & R Winders. London: Kogan Page.

[Boehm 1988] Boehm B W. 1988. A spiral model of software development and enhancement. *IEEE Computer*, **21**(5), 61 – 72..

[Booch 1994] Booch G. 1994. *Object-Oriented Analysis and Design: with Applications* (2nd ed.) Redwood City, CA: Benjamin/Cummings Publishing Company, Inc.

[Budgen 1994] Budgen D. 1994. *Software Design*. Wokingham: Addison-Wesley.

[Chen 1989] Chen J W & Shen C. 1989. Software engineering: a new component for instructional software development. *Educational Technology*, **29**(9), 9 – 15.

[Coad 1995] Coad P, North D & Mayfield M. 1995. *Object Models: Strategies, Patterns & Applications.* Englewood Cliffs, NJ: Prentice Hall.

[Conger 1994] Conger S A. 1994. *The New Software Engineering.* Belmont, CA: Wadsworth Publishing Company.

[Dearden 1995] Dearden A M. 1995. *The use of Formal Models in the Design of Interactive Case Memory Systems.* DPhil Thesis, University of York (UK).

[Dix 1991] Dix A. 1991. *Formal Methods for Interactive Systems.* London: Academic Press.

[Dix 1993] Dix A, Finlay J, Abowd G & Beale R. *Human-Computer Interaction.* Hemel Hempstead: Prentice-Hall.

[Gray 1993] Gray D E & Black T R. 1994. Prototyping of computer-based training materials. *Computers in Education*, **22**(3), 251 – 256.

[Harrison 1990] Harrison M D & Thimbleby H (Eds). 1990. Formal Methods in Human-Computer Interaction. Cambridge: Cambridge University Press.

[Henderson 1990] Henderson-Sellers B & Edwards J M. 1990. The Object-Oriented Systems Life Cycle. *Communications of the ACM*, **33** (9), 142 – 159.

[Johnson 1988] Johnson R & Foote B. 1988. Designing reusable classes. *Object-Oriented Programming*, **1**(2), 22 – 35.

[Korson 1990] Korson T & McGregor J D. 1990. Understanding object-oriented: a unifying paradigm. *Communications of the ACM*, **33** (9), 40 – 60.

[Lantz no date] Lantz K E. (no date). *The Prototyping Methodology.* Englewood Cliffs, NJ: Prentice-Hall.

[Merrill 1988] Merrill M D. 1988. Applying component display theory to the design of courseware. In: *Instructional Designs for Microcomputer Courseware*, edited by D H Jonassen. Hillsdale, NJ: Lawrence Erlbaum Associates.

[Palanque 1995] Palanque P & Bastide R (Eds). 1995. *Proceedings of the Eurographics Workshop in Toulouse France, June 1995.* Wien: Springer-Verlag.

[Paternó 1995] Paternó F (Ed). 1995. *Interactive Systems: Design, Specification and Verification.* Berlin: Springer-Verlag.

[Schach 1996] Schach S R. 1996. *Classical and Object-Oriented Software Engineering* (3rd ed.). Boston, MA: Aksen Associates Inc. Publishers.

[Shneiderman 1992] Shneiderman B. 1992. *Designing the User Interface.* Reading, MA: Addison-Wesley.

[Sommerville 1992] Sommerville I. 1992. *Software Engineering* (4th ed.). Wokingham: Addison-Wesley Publishing Company.

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

[Tracz 1988] Tracz W (Ed). 1988. *Software Reuse: Emerging Technology.* Washington DC: IEEE Computer Society Press.

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

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