Current Issues in IT Education

edited by

Tanya McGill
Murdoch University, Australia

IRM Press
Publisher of innovative scholarly and professional information technology titles in the cyberage
Hershey • London • Melbourne • Singapore • Beijing

Acquisitions Editor:	Mehdi Khosrow-Pour
Senior Managing Editor:	Jan Travers
Managing Editor:	Amanda Appicello
Copy Editor:	Lori Eby
Typesetter:	Amanda Lutz
Cover Design:	Weston Pritts
Printed at:	Integrated Book Technology

Published in the United States of America by
IRM Press (an imprint of Idea Group Inc)
701 E. Chocolate Avenue, Suite 200
Hershey PA 17033-1240
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@idea-group.com
Web site: http://www.irm-press.com

and in the United Kingdom by
IRM Press (an imprint of Idea Group Inc)
3 Henrietta Street
Covent Garden
London WC2E 8LU
Tel: 44 20 7240 0856
Fax: 44 20 7379 3313
Web site: http://www.eurospan.co.uk

Library of Congress Cataloging-in-Publication Data

McGill, Tanya.
Current issues in IT education / Tanya McGill.
p. cm.
Includes bibliographical references and index.
ISBN 1-931777-53-5 (soft cover) -- ISBN 1-931777-69-1 (ebook)
1. Information technology--Study and teaching. I. Title.
T58.5.M38 2003
004'.071--dc21

2002156239

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

Chapter VIII

Conceiving Architectural Aspects for Quality Software Education through the Constructivist Perspective

Kam Hou Vat
University of Macau, Macau

ABSTRACT

This chapter describes the initiative to incorporate the practice of quality software education (QSE) into our undergraduate curriculum concerning the engineering of software. We discuss how the constructivist's method of problem-based learning (PBL) helps develop this QSE practice in our students' daily learning. We also expound the idea of an architectural context to building information systems (IS) solutions, supported by the industry's emerging consensus that architecture provides the kind of thinking and methods we need to develop today's complex systems. Our QSE approach focuses on designing problems, which require the building of a sensible IS architecture characterized by objects of different services. Our QSE approach is outlined in terms of a state-of-the-practice management philosophy called action learning, modified for educational scenarios, so that our students could learn to acquire their collaborative software engineering and management experience in the practice of architected applications development. To conclude, the criteria used to evaluate the working of our learning scenario and the challenge in combining action learning with PBL in innovating different QSE experiences for our students is discussed.

INTRODUCTION

In today's knowledge economy (OECD, 1996), as the possibilities of the information revolution challenge traditional business logic, many an organization has embarked on the journey of electronic transformation (Cook, 1996; Umar, 1997). According to Hammer and Champy (1993), this is the fundamental rethinking and radical redesign of business processes to achieve dramatic organizational improvements. In the past, the order of the day was to reorganize the technology each time a business changed. Yet, the reoriented consensus was to facilitate software solutions that adapted as the business adapted. This support for an increasingly adaptive business is currently achieved through the reuse of business components (Eeles, 2000), which are executable units of code that provide physical black-box encapsulation of related business services, accessed through a consistent, published interface that includes an interaction standard with other components. These business components support a process-based view of the business as it changes. Consequently, it is important to derive the necessary business models, which are traceable back to the originating requirements, in order to provide a secure foundation to develop the component-based information systems (IS) support. Indeed, this is often the backdrop behind which most of our universities' undergraduate programs in Software Engineering and Information Systems have been running. The fact of the matter is, we are often confronted with the situation (Dawson & Newsham, 1997) that most of our graduates today begin their careers lacking an appreciation of real-world conditions (Speed, 1999; Wasserman, 1996; Shaw, 1990). As academics, the haunting question is this: How do we cultivate future graduates who become more prepared to tackle real-world problems in the engineering of software for quality IS solutions, starting from their university education? This chapter serves as an educational response to devise suitable quality software education (QSE) scenarios for our students' active learning experiences. In the following discussion, we first introduce our architectural context for IS education, then provide a briefing on our pedagogy of action learning (Dean, 1998; Dilworth, 1998b; Revans, 1998) substantiated with problem-based learning (PBL) (Albanese & Mitchell, 1993; Engel, 1991; Ryan, 1993; Barrows, 1985). Next, we present some scenarios for enterprise's electronic transformation pivoted by e-business initiatives, followed by our elaboration of some architectural topics of our QSE curriculum. Finally, we discuss our criteria in evaluating the practice of PBL as well as some lessons learned from conducting the QSE.

THE ARCHITECTURAL CONTEXT FOR IS EDUCATION

Our discussion of the architectural context for IS education is centered about several themes: first, to clarify why we need architecture to build IS solution; second, to define what constitutes architecture in the IS context; and third, to provide a high-level introduction to the architectural approach to building IS solutions.

The WHY of Architecture in IS

The key technical issue in developing an information system — be it a conventional IS or a Web IS — is why we need an architecture in IS construction. We could resort to the insight and intuition of a building architect to extrapolate to the IS world and propose a list of

requirements to be fulfilled by our architecture in the context of IS solution building. Essentially, the function of a building architect can be summarized as follows (Buffam, 2000; McConnell & Tripp, 1999):

> *The architect creates in his or her mind a concept of the overall form of the building to fit the intended purpose. This same architect creates a tangible set of blueprints that express his or her concept with sufficient clarity and rigor that the building owners can verify that the design satisfies their needs. Also, the architect — before committing to construction — can verify, through inspection, simulation, and calculation, that the building will stand up to its anticipated load, withstand environmental conditions and requirements, and meet regulatory standards. Tellingly, craftsmen can construct a building fulfilling that concept.*

Accordingly, in the IS context, we could provide a number of reasons to support the provision of an architecture. First, we need this architecture to ensure that the IS environment is aligned with the organization's imperatives. Namely, this architecture provides the basis for IS professionals and organizational leaders to ensure that the proposed system is properly aligned with the mission, objectives, and processes of their business. Such an alignment supports typical organizational goals as enhancing the capabilities of existent information systems and taking advantage of new strategic opportunities. Second, we need the architecture to help build an IS environment that can be easily changed and extended, so as to retain its alignment with changing business imperatives in the organization. Third, we need architecture to communicate appropriate views of the solution to, and among, the various stakeholders, so as to ensure that the solution gets built on time and within budget, while fulfilling the intended requirements. Fourth, we need architecture to help keep our IS environment (and its supporting processes) intellectually manageable. We recognize that information systems are complex. The control of complexity, and through it the ability to keep our systems understandable, is the biggest single challenge in the IS construction. One of the most important functions of the architecture is to support a "divide-and-conquer" approach. Other functions include to provide a framework for making and communicating technology choices, to give us freedom of choice of information technology (IT) components through component interoperability and through component portability, and to maximize our efficiency in building and evolving the IT environment through reuse of earlier work. In other words, it is too important for IS/IT professionals to neglect the essence of architecture — the reminder of a whole sequence of organizational and technological concerns.

The WHAT of Architecture in IS

The architectural context of IS solution building could be considered as a set of principles acting on and intimately integrated with, the total process of creating IT solutions. This process is often formulated in several distinct directions, such as the common-component sense, the design sense, the blueprinting sense and the framework sense (Pour, Griss, & Lutz, 2000; Bourque et al., 1999; Repenning et al., 2001; Zachman, 1987):

- The *Common-Component Sense*: This sense is based on the idea of reusability; namely, design is based on leveraging reusable standard components, subassemblies, frameworks, patterns, and idioms. To understand its significance, we can compare a traditional IS design with one guided by architectural principles. Traditional IS design

involves such activities as understanding the business domain, abstracting models for this domain, and crafting application components to realize the models. Often, we attempt to excavate reusable components from previously developed systems. In contrast, the architectural way of IS design involves the following: understand the business domain, match the business domain to standardized architectural models, and adapt the components associated with these models to meet domain requirements.

- The *Design Sense*: This sense is based on a number of requirements for architecture to ensure that the IT environment is aligned with business imperatives. First is the mission of designing a solution to meet a client's needs. Second is the conscious imposition of principles and guidelines into the design activity, governing the structure of design. Third is the formulation of standards to be observed in implementing the design. Fourth is the activity dealing with the higher levels of abstraction in design. In this sense, what is important is the discipline we bring to the design process, the principles and guidelines that impose order so as to shape and constrain the design in ways that will ensure its ultimate success. To achieve elegant designs, as opposed to those that are merely adequate, the software architect's challenge is to create systems that are in perfect harmony with their intended purpose. The word "elegance" captures this quality most aptly, because it represents a clear, intuitive mapping between a function and its implementation. Elegance is desirable, because it brings intellectual manageability in the design activity.
- The *Blueprinting Sense*: The blueprinting sense of the word architecture is to produce blueprints that are comprehensible at appropriate levels of abstraction, to fulfill the needs of different stakeholders viewing the system from different angles. In current practice, the blueprinting function is effectively integrated into the modeling activity. We model the business, and we model the information systems that support its business processes. The methods that we use in these modeling activities incorporate the blueprinting function.
- The *Framework Sense*: The framework sense denotes a finished design of some kind. Where architecture in the finished-design sense is helpful, is where we can abstract some more generalized, or completely domain-independent, behavior that can serve as a framework for other solutions. The word architecture used in this sense is supportive of the common-component sense. Namely, by applying architectural principles in our solution building, we tend to produce designs that reuse proven frameworks.

The HOW of Architecture in IS

Following our discussion of a set of requirements that the word "architecture" has to fulfill, and a set of directions commonly attached to this word, we could conceptualize the architectural way to IS solution building as follows:

- *Targeting for client needs*: This is the most important characteristic of the architectural approach; we must design a solution to fit our client's needs.
- *Using validated principles*: The architectural approach conducts design according to vital principles that have been found to be common to successful systems. Examples include a clear separation of concerns between interface and implementation, and construction based on a hierarchy of well-defined layers of abstraction.

- *Reusing components, patterns, and frameworks*: As far as possible, we assemble our systems from available prebuilt components, in commonly understood and well-recognized patterns, structured around familiar frameworks.
- *Achieving elegance in all endeavors*: We strive for the elegant solution, for the simple and obvious. We should adhere to implementation principles covering any topic required to provide the proper guidance in decision making, including those for technology selection and for requirements governing nonfunctional attributes of the system to be built.
- *Adopting formal description for records*: We should use a formal description and recording discipline that represents the requirements for the IS system and its functional and environmental characteristics at various levels of abstraction. All the stakeholders in the system can relate to one or more representations of the system specification to verify that their needs are being fulfilled and that they understand how to advance the realization of the system to the next level of refinement.

All of these ideas are meant to help develop in our students their abilities in handling the software challenges of the Internet age. These include the following (Allen & Frost, 1998; Lethbridge, 2000; Meyer, 2001): support increasingly adaptive businesses, capitalize on the rapid advances in component technology; deal with legacy systems; plan and build for reuse; prepare for quality issues; and retain a pragmatic focus in the face of increasing complexity. Collectively, these challenges represent the drivers of change, worthy enough to secure a place in our discussion of QSE.

THE ACTION LEARNING MODEL OF EMPOWERMENT

To facilitate students' involvement of IS design and construction, and to understand the way organizations learn to improve themselves, we suggest adopting the discipline of action learning (PIQ, 1998) as an empowerment companion in the students' excursion of electronic transformation among enterprises. We interpret action learning (Dean, 1998) as a voluntary, participant-centered, evolutionary process to solve real, systemic, and pending organizational problems in the workplace. Its central mission (Dilworth, 1998b) is to increase the capacity of individual learners and the learning of the organizations they are associated with, to adapt to a rapidly changing environment. Revans (1998), who is widely known as the principal pioneer of action learning, suggested that it is eclectic, cutting across many fields. It emphasizes action, reflection, the need for critical thinking, and a climate of trust and authenticity. In action learning, the learning process is fueled by real problem solving among the participants. Its basic learning model can be characterized as follows:

> *A number of managers get together at regular intervals to discuss a problem or challenges they are facing in the workplace. The group referred to as the set in action learning literature, usually has a resource person, though the role of this person changes from context to context. After discussing the problem, project, or challenge with the set and the resource person, the managers return to the workplace to take action. After a period of time, the set meets again to*

discuss progress to date, results achieved, and problems still to be resolved. The managers then return to the workplace to take further action. The two phases of reflection (discussion) and action continue to alternate throughout the life of the learning period.

The Pedagogy of Problem-Based Learning

To translate Revan's description of action learning into terms applicable to the IS students' exploration, we have chosen to substantiate the action-learning context with problem-based learning (PBL) activities (Savery & Duffy, 1995; Albanese & Mitchell, 1993; Engel, 1991; Ryan, 1993) as follows:

1. Students, divided into small groups of three to five members and assigned a facilitator, are encouraged to perceive themselves as managers of their own in terms of time, material resources, and complexity of the problems that can be handled one at a time by the group.
2. Students are made aware that initially they will not possess enough prior information to solve the problem at hand or to clarify the scenario immediately.
3. Students are challenged to construct a solution to an often ill-structured problem chosen according to some concrete, open-ended situations.
4. Students are reminded that they must identify, locate, and use appropriate resources, and ask questions referred to as learning issues on the various aspects of the problem. These learning issues help the IS students realize what knowledge they require, and thus focus their learning efforts and establish a means for integrating the information they acquire.

It is expected that the IS students' groups generally have to iterate through some relevant stages of activities: analysis, research, and reporting, with discussion and feedback from peers and the facilitator at each stage:

- *Analysis*: Throughout this stage, students organize their ideas and prior knowledge related to the problem, and start defining its requirements. This helps them devise a specific statement of the problem. Meanwhile, they are urged to pose learning issues, defining what they know and what they have to know. This helps them assign responsibilities for research, eliciting and activating their existing knowledge as a crucial step in learning new information.
- *Research*: Throughout this stage, students collect necessary information on specific learning issues raised by the group. They may conduct library searches, seek sources on the Internet, collect data, and interview knowledgeable authorities. More importantly, when they come to realize the complexity and texture of the problem, they become their own experts to teach one another in the group; they use their learning to reexamine the problem. In the process, they are constructing knowledge by anchoring their new findings on their existing knowledge base.
- *Reporting*: After a specified period of time, students reconvene and reassess the problem based on their newly acquired knowledge. Once the students feel that the problem task has been successfully completed, they discuss the problem in relation to similar and dissimilar problems in order to form generalizations. Meanwhile, the facilitator's feedback should help students clarify basic information, focus their

investigations, and refine their problem-solving strategies, besides addressing whether the original learning issues were resolved and whether the students' understanding of the basic principles, information, and relationships is sufficiently deep and accurate.

The Unifying Formula of Action Learning

A frequent formula (Dilworth, 1998a) that action learning proposes is L=P+Q+R, where Learning (L) equals Programmed Instruction (P) plus Questioning (Q) plus Reflecting (R). Here, P represents the knowledge coming through textbooks, lectures, case studies, computer-based instructions, and others. This is an important source of learning but carries with it an embedded caution flag. Namely, P is based in the past. Q means continuously seeking fresh insight into what is not yet known. This Q helps avoid the pitfall of imperfectly constructed past knowledge. By going through the Q step first, we are able to determine whether the information available is relevant and adequate to our needs. It will point to areas that will require the creation of new P. R simply means rethinking, taking apart, putting together, making sense of facts, and attempting to understand the problem. Following the use of this formula, action steps are planned and carried out with constant feedback and reflection as the implementation takes place. In short, what action learning can provide for the IS student-groups is elevated levels of discernment and understanding through interweaving action and reflection.

THE LEARNING SCENARIO FOR QSE

It is understood that collaborative project work (Favela & Pena-Mora, 2001) is recognized as having many educational and social benefits (Wills, 1998), in particular, providing students with opportunities for active involvement with their study. However, teaching, directing, and managing group-based project work is not an easy process. This is because projects are often expensive, demanding considerable supervision and technical resources; and complex, combining design, human communication, human–computer interaction, and technology to satisfy objectives ranging from consolidation of technical skills through provoking insight into organizational practice, teamwork, and professional issues, to inculcating academic discipline and presentation skills. More tellingly, PBL as a process-oriented instructional method helps prepare our students to get started with group project work to initiate their immediate journey as future IS/IT professionals in software development. Our learning scenario for QSE, based on real-world findings, is designed incrementally to arouse students' attention to different areas of concerns in the electronic transformation of today's enterprises.

The Demand of e-Transformation

It is increasingly obvious that e-business (Amor, 2000), conducted in and around the global marketplace, has presently become one of the most important drivers for electronic transformation (e-transformation) of today's enterprises. Yet, it has been commented that the long-term potential of e-business requires prudent contemplation and planning on the part of management. The formulation and implementation of e-business strategies, applications, and services involves many business issues that the traditional IS/IT department could not handle on its own (Kalakota & Whinston, 1996). Instead, the emerging consensus is to

develop a cross-functional team composed of technical staff as well as business architects who may not know much about technology but who understand the core business. It is believed that such teams could integrate efforts and streamline cooperation among different functional departments to create business processes that are efficient, effective, and responsive.

The New Trade Model in e-Business

With the emergence of the Web-based e-business, enterprises today need a new model for trade that addresses new requirements in the Internet economy. It is found that the term "dynamic trade" (Leif, 1998) is often used to define an enterprise ability to satisfy current demand with customized response. Dynamic trade is expected to go beyond today's Web efforts, which extend traditional trade with easily available online data, and with customer self-service for simple inquiries like reviewing account history or checking order status. Instead, dynamic trade is meant to enable companies to maximize the lifetime value of a business relationship, through such value-added services as creating product and service bundles based on actual consumer preferences, and using traditional data to react to market changes.

The New Role of IS/IT in e-Business

It has been commented (McCarthy, 1999) that traditional IS/IT departments have largely adopted an inward-looking perspective to cater to only the internal users of an enterprise, but organizations must now work with a new externally focused business model like dynamic trade. This will inevitably present challenges to the internally focused IS/IT, often preoccupied with such goals as cutting costs and reducing risks. As Internet economy develops, the emphasis will shift from dumping product information onto the Web pages to delivering customized services, like buying assistance for consumers or proactive inventory management for business partners. According to Cameron (1999), one of the new roles for IS/IT departments is to become involved in developing new business software that will help companies exploit the promise of dynamic trade by enabling firms to capture information, analyze it, and respond to customers in real time. Meanwhile, IS/IT technology personnel will need more integration specialists, project managers, and business liaisons to ensure that business processes flow smoothly across internal and external boundaries of the enterprise.

The Enabling Technologies Behind e-Business

One of the main enabling technologies behind e-business development is the reuse of software components over some standardized distributed-object middleware (Berstein, 1996), through which such components can be moved around at execution time and deployed in a way that optimizes the technology in order to deliver the most business benefit. These advances in component technology have resulted in the movement toward separation of software applications from the increasingly heterogeneous technology platforms on which the services are deployed (Anderson & Dyson, 2000; Cook, 2000; Braude, 2001). It provides the potential for an application to be physically distributed so that it services the needs of the business and not the technology. We call this the service-based view of software construction, where components provide a method of packaging related services into prefabricated pieces of software from which solutions can be constructed. This service-based approach is also applicable in the area of legacy software, where most development

is about enhancing existing systems, providing new front-ends to established back-ends, capitalizing on existing relational technology for data storage, and building interfaces to existing packages. It allows organizations to wrap the existing services into new offerings or products, so as to reuse their investments in existing packages, databases, and legacy systems within the context of component technology.

The Critical Problems Underlying e-Business Development

The overall picture confronting enterprises today could be characterized as this (Cook, 1996, 2000): at the core is the installed base of existing IT systems, which includes the legacy data and business logic. Around the edge are increasingly proactive customers, to which the enterprise must offer an increasing quality of service through existing and new channels. In between, the enterprise is reengineering its business processes, with a focus on knowing its customers better, and offering continuous improvement of its products and services. From an IT perspective, the legacy systems become surrounded by a matrix of go-between componentry providing services to support the changing business, with increased flexibility and reduced development times as compared with legacy systems. This is often a challenge requiring skilled and thorough design, taking into account such attributes as reliability, efficiency, usability, maintainability, testability, portability, and the most essential reusability. Nonetheless, a common reaction to the pressure of immediate business needs is to virtually abandon planning and control in the name of producing fast results (Gartner Group, 1995). Yet, incremental releases that are developed in isolation solely to meet tight deadlines will eventually result in fragmented systems that lack consistency and fail to provide integrated support. Worse still, this presents a problem that grows out of control exponentially with the number of increments delivered.

The Architectural Way to e-Business Solution Building

We believe that a key requirement of an incremental approach to e-business solution building is to base increments on a sound architecture (Boehm & Basili, 2000) that enables components to be plugged in as service providers to the increments. Besides, this should be an architecture for model building (Zachman, 1987), which supports such goals as management of scale and complexity, interoperability, and adaptability. In large organizations with complex business processes, there is a need to manage scale and complexity in software development in such a way that the resulting software structure mirrors business needs as closely as possible (Gartner Group, 1996). This requires the architecture to establish the definitions, rules, and relationships that will form the infrastructure of models from business process to code. More, this architecture should support the idea of software evolution among a mix of legacy systems and databases, off-the-shelf packages, and newer applications.

ARCHITECTURAL MODELING BASICS FOR QSE

To help our PBL students embark on their journey of system development, we selected some recurring architectural modeling concepts for them to learn and practice through trial, error, and mentoring. These concepts are based on fundamental principles (Hartley, Hruschka, & Pirbhai, 2000) that we believe are applicable to IS modeling, regardless of the underlying techniques used.

The Layer-System Concept

We believe each system is a component of one or more larger systems. The larger systems are the context or environment in which the component system must work. Systems comprise, thus, a layered set of subsystems below the layer with which we happen to be dealing and a layered set of supersystems above that layer. This layered structure can be exploited in representing systems and in defining the system development process. Most systems are members of multiple-layered sets. The particular set(s) chosen to represent a system are determined by the viewpoint(s) that are important for the particular system. Also, every system has a set of essential requirements, which meet the needs of the context or environment, and a set of physical requirements, which reflect the architectural and design decisions made to satisfy the essential requirements. To succeed in the development of complex systems, all system artifacts invoked by these principles must be represented separately, but their relationships and interactions must also be represented.

The Modeling Concept

A model is an abstraction highlighting some aspects of real-world systems in order to depict those aspects more clearly. A model has an objective (the question we want it to answer) and a viewpoint—the point of view of one or more stakeholder(s). Abstract models reduce the complexity of the real world to digestible chunks that are simpler to understand. Different types of models answer different types of questions about the system they represent. If we decide to build more than one model of a given system to investigate different aspects, then we should somehow organize these models according to their relationships to one another and to the system. Hence, we often need a framework to accommodate different models.

The Separation of Concerns Concept

Every system has a specification comprising two important parts: system requirements and system architecture. Both of these parts contain models. The "system requirements" model is a technology-independent model of the problem the system is to solve. It represents the "what." The "system architecture" model is a technology-dependent model of the solution to the problem. It represents the "how." Typically, these two models are created for the entire system and for every subsystem down to the lowest level in the system hierarchy. And, it is important to separate the "what" and the "how" for the following reasons: It is often useful to understand a problem independently of any particular solution. Any given problem has many possible solutions. Selection of a particular solution is a trade-off process; we often need to make numerous different trade-offs while keeping the problem statement unchanged. The separation should support the principle of separation of concerns, which means dealing with only one part of the system's complexity at a time. The "requirements" model has to cope with only the essential problems; the "architecture" model has to cope with many constraints imposed by technology and organization. This separation of the "what" and the "how" gives us the flexibility to re-implement the "what" using new technology, but it also gives us the convenience of reusability — not just for software or hardware but for requirements as well. This is particularly important, because requirements are relatively more stable over longer periods of time than technology.

The Major Modeling Relationships for Layer-System

Four types of modeling relationships have been of particular interest, which are distinctly different from one another. They are, respectively, the relationships of aggregation-decomposition, abstracting-detailing, supertype-subtype and controlling-controlled. They all serve distinct and important roles in system development. They can be integrated smoothly, where appropriate, with other models, including object-oriented models. The following provides features of each relationship:

- *Aggregation/decomposition relationship*: Through this relationship, elements in the higher layers actually consist of the elements in the lower layers, or conversely, elements in the lower layers are decompositions of those in the higher layers. This structure is also known as a whole/part structure or a container/content structure: namely, a given layer provides the container for the layer below, which is the content of the layer above. In practice, an aggregate involves more than just collecting sub-elements into a set. The sub-elements must also interface with one another, requiring linkages between them that may not be evident when they are considered separately. When applied to software, we can imagine an architecture module at the highest software layer; major subprograms it contains are modules in the next layer down; sub-subprograms or subroutines form a further layer.
- *Abstraction/detailing relationship*: Through this relationship, the higher layers are simply more abstract expressions of the lower layers, or conversely, the lower layers are more detailed expressions of the higher layers. It is important to notice this. An abstract requirement statement does not contain the more detailed requirements statements that describe it. If we assemble a set of detailed requirements, we merely have a collection of detailed requirements — the abstract and detailed requirements exist independently of each other, with an abstraction/detailing relationship between them.
- *Supertype/subtype relationship*: Through this relationship, an element in the higher layer — the supertype — includes all of the features that are common to its associated elements in the lower layer — its subtypes. These features are attributes that are inherited by the elements on the lower layer. Starting from the lower level, supertypes are formed for sets of elements that share common attributes. Supertype/subtype models are important in object orientation. This relationship is the foundation for inheritance. Moreover, object orientation has taken this relationship and extended it to more complex forms of inheritance than just attribute inheritance: The lower layer may also inherit functions and the behavior of the supertypes. With the supertype/subtype relationship, it is important that the supertype contain all the commonalities of the subtypes. The main use of this relationship is to discover commonalities and to describe them only once, thus reducing redundancy. The structure then allows the lower layers to inherit whatever commonalities have been discovered. We can see that this relationship is a subtype of the abstraction/detailing relationship. A supertype is an abstraction of its subtypes, and the subtypes are detailed instances of the supertype. Other names used for this relationship include generalization/specialization, class hierarchies, inheritance structures, and "is-a" hierarchies.
- *Controlling/controlled relationship*: Through this relationship, the upper layers control elements of the lower layers. Other terms used for this relationship are control hierarchy, or the is-boss-of (is-supervised-by) relationship. Sometimes, we simply say

that the higher element uses the lower elements. The higher layer must have knowledge of the lower layer, but the lower layer — that is, the one being used — does not necessarily have to know anything about the boss. In terms of client/server models, the client is the boss that delegates work to the server; the server provides certain services that are performed whenever a client asks for them.

Typically, layered models based on the four types of relationships can be used simultaneously to represent different aspects of a system. For example, the required functional capabilities of a system can be captured by a process model, which is based on the abstraction/detailing relationship. The required behavioral capabilities are captured by a control model, which is based on a controlling/controlled relationship. The information structures in the system are captured in an entity-relationship model based on the supertype/subtype relationship. Also, the physical structure can be captured by the architecture model, which is based on an aggregation/decomposition relationship. In sum, the layered models allow us to represent different views of the system separately, but when done as part of the requirements and architecture models, the links between these views are carefully maintained.

THE COMPONENT-BASED IS ARCHITECTURE AND PROCESS

Our study of component-based development (CBD), based on the Allen and Frost (1998) model, is evolutionary in nature. We aim to harness a service-based method with effective object-oriented modeling to capitalize on the increasing power of the fast-developing component technology. The idea is to provide an overall design philosophy for realizing the vision of service-based reuse of components (Allen & Frost, 1998; Anderson & Dyson, 2000; Cook, 2000). We call this philosophy the *service-based architecture* for CBD, which employs the concept of *service packages* to facilitate a business-oriented modeling process. A service package provides a set of services belonging to a single service category. Each service from a service package is realized by an individual component, which is also a container of different objects. This provides a business-oriented basis for modeling deployment of components using *services packages*, which are implementation packages of objects, providing services through their interfaces. That way, components provide a means of packaging related objects together into prefabricated pieces of software. And, service packages provide a mechanism for grouping those objects into units (in the form of components) that are cohesive to the needs of a particular set of services from which business solutions can be constructed. It is important to notice that the promise of component-based development is that software solutions can be composed from reusable components, in analogous fashion to hardware (Cox, 1986; Eeles, 2000; Repenning et al., 2001). Nevertheless, the service packages must be modeled in a way that makes the resulting components useful building blocks, simple to activate and inexpensive to administer. The level of granularity of a component can vary from large and complex to small and simple. In practice, large components have the greatest potential for reuse but are often not cohesive and may be difficult to assemble into solutions with other components. Small components are usually more cohesive but often need to be coupled with many other components to achieve significant reuse, resulting in excessive intercomponent coupling. Clearly, settling on a good and useful level of granularity is a trade-off between these two extremes.

The term "process" as used in our pedagogic context for architected applications development (AAD), carries the connotation of process models designed to view the real world from the viewpoint of architectural software development. Thus, the process to be described is an abstract description of the software development activities within the service-based architecture (Stapleton, 1997). We are interested in a two-tier process to achieve an AAD methodology: the solution process, and the component process. The former is aimed at development of solutions, typically in terms of user services, to maximize reuse of existing services and provide early user value. The latter is aimed at developing components that provide commonly used business and data services across different departmental systems or for use by third parties. It is important to notice that we often need to use elements of both processes adapted to our specific needs. Typically, the key driver of the solution process is a set of specific requirements to meet the needs of a business process. Various models are produced throughout the process, which evolve in detail as the process unfolds. We continually seek opportunities to extend and refine existing generic models. Such models are often selected on a use-case by use-case basis (Jacobson et al., 1992) for incremental development of user services. On the other hand, the generic business requirements that drive the component process may come from the need to reuse existing legacy assets, and the feedback from solution projects. An important part of the component process is to evolve the models so that they can be specialized and refined by solution builders to form an evolving set of components.

In practice, the use of a process by a software development team should assist project management in numerous tasks. These include identification and partitioning of work, identification of progress achieved, planning of the staff resource profile, planning of the requirement for physical resources, and provision of cost and time scale estimates for the work yet to be performed. From a technical viewpoint, a process should assist in such areas as identification of preconditions required before each activity is started, specification of the products and deliverables required from each activity, techniques that may be used during each activity, and experience gained from earlier work. Clearly, building and refining generic models is an important aspect of CBD, where we want to leverage model reuse more than code reuse. Service packages provide a means of structuring a project in terms of architectural context and allow us to build on and capitalize on the best work of others. A service package can be effectively employed in a component process to contain a generic model, which can be refined and extended to meet the specific needs of a solution process. The model solution space evolves to contain more detail as a project moves through the iterations of the process. Eventually, portions of the model are mature enough to be transformed into code. The tested code represents the model at its most detailed level of abstraction. As for deliverables, they are simply views of a maturing model. Indeed, the service-based process for AAD is an adaptive process that can be tuned and customized to specific organizational needs. Checkpoints can also be built into the process to help evolve it. This includes documenting the lessons learned so that others can avoid making the same mistakes.

TEAM SKILLS DEVELOPMENT FOR MANAGING SOFTWARE REQUIREMENTS

To partially address the requirements challenge in architectural IS solution development, we often suggest some team-based activities (DeMarco, 1982; Jacobson et al., 1992;

Leffingwell & Widrig, 2000; Brown & Dobbie, 1998) to be experienced by our PBL students within their curriculum activities:

- *Analyzing the problem*: This includes a set of skills to understand the problem to be solved before application development begins. It is the process of understanding real-world problems and user needs and proposing solutions to meet those needs. We consider a problem as the difference between things as perceived and things as derived (Gause & Weinberg, 1989). Accordingly, if the user perceives something as a problem, it is a real problem, and it is worthy of addressing. Typical techniques include gaining agreement on the problem definition, understanding the root causes to induce the problem, and identifying the stakeholders and the users, with the former being anyone who could be materially affected by the implementation of the new application.
- *Understanding user needs*: This introduces a variety of techniques to elicit requirements from the system users and the stakeholders. Software teams are rarely given effective requirements specifications for the systems they are going to build. Often, they have to go out and get the information they need to be successful. Typical methods include interviewing and questionnaires, requirements workshop, brainstorming and idea reduction, storyboarding, use cases derivation, role playing, and prototyping. Each represents a proactive means of pushing knowledge of user needs forward and thereby converting fuzzy requirements to those that are better known.
- *Defining the system*: This describes the initial process, by which the team converts an understanding of the problem and the users' needs to the initial definition of a system or application that will address those needs. Our PBL teams should learn that complex systems require comprehensive strategies to organize information for requirements. This information could be expressed in terms of a hierarchy, starting with user needs, transitioning through feature sets, then into the more detailed software requirements. The latter could be expressed in use cases or traditional forms of requirements documents, say, the vision document defining at a high level of abstraction, both the problem space and the solution space.
- *Managing the project scope*: This reminds our teams that they should be aware not to initiate projects with too large a scope to be accomplished. Project scope is presented as a combination of the functionality to be delivered to meet users' needs, the resources available for the project, and the time allowed in which to achieve the implementation. The purpose of scope management is to establish a high-level requirements baseline for the project. The team has to establish the rough level of effort required for each feature of the baseline, including risk estimation on whether implementing it will cause an adverse impact on the schedule. Also, each team has to actively engage its customers in helping solve the scope management problem to ensure the quality and the timeliness of the software outcomes.

THE CRITERIA FOR PBL EVALUATION

Throughout our students' study period, we have borne in mind that our instructional method should be evaluated in part by its ability to explain practice. The following explicit criteria (Greening, 1998; Ryan, 1993; Savery & Duffy, 1995) have been found useful in order to later judge the learning outcome with respect to the process of problem diagnosis, action intervention, and reflective learning:

- *Learning is an active and engaged process*: Instead of being told what to do or how to solve problems, students within a PBL atmosphere are to generate their own learning issues. It is expected that a sense of ownership should be born, leading to greater cognitive engagement. Students are actively engaged in working at tasks situated in an authentic setting, which should lead to greater ability in transfer to other real-world contexts.
- *Learning is a process of knowledge construction*: PBL purports that learners construct their own knowledge. The constructivist epistemology states that the known is internal to the knower and is subjectively constructed based on individual responses to experience. Thus, in order to harness the reality of learning, we need to consider the opportunity to find knowledge for oneself, contrast one's understanding of that knowledge with others' understanding, and refine or restructure knowledge as more relevant experience is gained.
- *Learners function at a metacognitive level*: Constructivist learning focuses on initiative thinking activities rather than working on the "right answer the teacher wants." Students generate their own strategies for problem formulation and possible solutions. The instructor's role is that of a facilitator, a guide, or a coach, probing students' thinking, monitoring their activities, and generally keeping the process moving. Thus, PBL should promote metacognition through encouraging students to reflect upon the problem-solving process. It is believed that reflection on recent experiences is an effective method of learning.
- *Learning involves social negotiation*: We accept the constructivist perspective that knowledge is socially negotiated. The quality or depth of one's understanding can only be determined in a social environment, where we can see if our understanding can accommodate the issues and views of others and to see if there are points of view that we could usefully incorporate into our understanding. A learning community, where ideas are discussed and understanding is enriched, is critical to the development of our students into self-directed work teams of software professionals.

REMARKS FOR CONTINUING CHALLENGE

It is experienced that the conventional approach to education remains the instructivist one, in which knowledge is perceived to flow from experts to novices. This transmissive view of learning is most evident in the emphasis on lectures, in the use of textbooks to prescribe reading, and in the nature of tutorials and assessment methods. It assumes that the process of good teaching is one of simplification of the truth in order to reduce student confusion. Yet, this simplification could deny students the opportunity to apply their learning to dynamic situations, such as quality software development through team-based collaboration. We question the transferability of the instructivist learning and ask how much of that which is assigned to academic learning ever gets applied to actual scenarios, when there is such a rapid surge in knowledge commonly associated with the birth of the "Information Age." This is a transference problem. Actually, the content product of learning is assuming a less important role relative to the process of learning, as the life of information content shortens, and the need for continual learning increases. In designing the learning scenario for QSE to be injected

into our project courses for software engineering and information systems, we tried to reorient toward a meaningful direction by reducing the obsession with knowledge reproduction. And, PBL represents one such relief from the constructivist pedagogy (Duffy & Jonassen, 1991). Greening (2000) described it as a vehicle for encouraging student ownership of the learning activities. There is an emphasis on contextualization of the learning scenario, providing a basis for later transference, and learning is accompanied by reflection as an important metacognitive exercise; for example, assessing how a project should be approached by an architectural context. Also, the implementation of PBL is done via group-based work, reflecting the constructivist focus on the value of negotiated meaning (Perkins, 1992). More importantly, it is unconfined by discipline boundaries, encouraging an integrative approach to learning, which is based on requirements of the problem as perceived by the learners.

On the other hand, when technology meets pedagogy, we insist that education of the architectural way to IS solution building should start with the ability to construct different models of interests, including the various business models and IS models. The result includes the design of a suitable IS architecture, denoting the integrated structural design of the system, its elements and their relationships depending on given system requirements. Conversely, this architecture has to represent all relevant aspects of a system, which are defined by models representing different system views. Such models are derived from the goals the system has to fulfill and the constraints imposed by the system's environment. From the standpoint of component-based development, we agree that our students should be given training to construct individual components efficiently. Then, their education should evolve through efficient development of component-based solutions in new domains, efficient adaptation of existing solutions to new problems, and efficient evolution of installed solutions by people with limited technical knowledge. Finally, it will achieve the efficient integration and evolution of sets of solutions. The real challenge is to derive a coherent set of architectural principles that will bring the whole of system development, including technology, methodology, and project management, into a single architecture-centric whole.

REFERENCES

Albanese, M., & Mitchell, S. (1993). Problem-based learning: A review of literature on its outcomes and implementation issues. *Academic Medicine, 68*(1), 52–81.

Allen, P., & Frost, S. (1998). *Component-Based Development for Enterprise Systems: Applying the SELECT Perspective*. Oxford: Cambridge University Press.

Amor, D. (2000). *The E-business (R)evolution*. New York: Prentice Hall.

Anderson, B., & Dyson, P. (2000). Reuse requires architecture. In L. Barroca, J. Hall, & P. Hall (Eds.), *Software Architectures: Advances and Applications* (pp. 87–99). Heidelberg: Springer-Verlag.

Barrows, H. (1985). *How to Design a Problem-Based Curriculum for the Pre-Clinical Years*. New York: Springer.

Berstein, P. (1996). Middleware: A model for distributed system services. *Communications of the ACM, 39*(2), 86–98.

Boehm, B., & Basili, V. R. (2000). Gaining intellectual control of software development. *IEEE Computer*, May, 27–33.

Bourque, P., Dupuis, R., Abran, A., Moore, J. W., & Tripp, L. (1999). The guide to the software engineering body of knowledge. *IEEE Software*, November–December, 35–44.

Braude, E. J. (2001). *Software Engineering: An Object-Oriented Perspective*. New York: John Wiley & Sons.

Brown, J., & Dobbie, G. (1998). Software engineers aren't born in teams: Supporting team processes in software engineering project courses. In *Proceedings of IEEE International Conference on Software Engineering: Education & Practice*, Dunedin, New Zealand, January, 26–29.

Buffam, W. J. (2000). *E-Business and IS Solutions: An Architectural Approach to Business Problems and Opportunities*. Reading, MA: Addison Wesley.

Cameron, B. (1999). Driving IT's externalization. January; www.forrester.com.

Cook, M. A. (1996). *Building Enterprise Information Architectures: Reengineering Information Systems*, New York: Prentice Hall.

Cook, S. (2000). Architectural standards, processes and patterns for enterprise systems. In L. Barroca, J. Hall, & P. Hall (Eds.), *Software Architectures: Advances and Applications* (pp. 179–190). Heidelberg: Springer-Verlag.

Cox, B. (1986). *Object-Oriented Programming: An Evolutionary Approach*. Reading, MA: Addison-Wesley.

Dawson, R., & Newsham, R. (1997). Introducing software engineers to the real world. *IEEE Software*, November, 37–43.

Dean, P. (1998). Editorial — Action learning and performance improvement. *Performance Improvement Quarterly, 11*(1), 3–4.

DeMarco, T. (1982). *Controlling Software Projects*. Englewood Cliffs, NJ: Yourdon Press.

Dilworth, R. L (1998a). Action learning in a nutshell. *Performance Improvement Quarterly, 11*(1), 28–43.

Dilworth, R. L. (1998b). Action learning — Setting the stage. *Performance Improvement Quarterly, 11*(1), 5–8.

Duffy, T. M., & Jonassen, D. H. (1991). Constructivism: New implications for instructional technology. *Educational Technology, 31*(5), 7–12.

Eeles, P. (2000). Business component development. In L. Barroca, J. Hall, & P. Hall (Eds.), *Software Architectures: Advances and Applications* (pp. 27–59). Heidelberg: Springer-Verlag.

Engel, J. (1991). Not just a method but a way of learning. In D. Bould & G. Felletti (Eds.), *The Challenge of Problem-Based Learning* (pp. 21–31). New York: St. Martin's Press.

Favela, J., & Pena-Mora, F. (2001). An experience in collaborative software engineering education. *IEEE Software, 18*(2), March/April, 47–53.

Gartner Group. (1995). Rapid application development, Part 2: Organizing for success. Inside Gartner Group This Week, June 7.

Gartner Group. (1996). Best practices in application development project management, Part 2, SPA-650-1293. ADM Research Note, March 20.

Gause, D., & Weinberg, G. (1989). *Exploring Requirements: Quality before Design*. Dorset House Publishing.

Greening, T. (1998). Scaffolding for success in problem-based learning. *Medical Education Online, 3*(4), 1–15, http://www.utmb.edu/meo/.

Greening, T. (2000). Emerging constructivist forces in computer science education: Shaping a new future? In T. Greening (Ed.), *Computer Science Education in the 21st Century* (pp. 47–80). New York: Springer.

Hammer, M., & Champy, J. (1993). *Reengineering the Corporation: A Manifesto for Business Revolution*. UK: Nicholas Brealey.

Hartley, D., Hruschka, P., & Pirbhai, I. (2000). *Process for System Architecture and Requirements Engineering*. Dorset House Publishing.

Jacobson, I., Christerson, M., Jonsson, P. M., & Overgaard, G. (1992). *Object-Oriented Software Engineering: A Use Case Driven Approach*. Reading, MA: Addison-Wesley.

Kalakota, R., & Whinston, A. B. (1996). *Electronic Commerce: A Manager's Guide* (pp. 1–29). Reading, MA: Addison Wesley.

Leffingwell, D., & Widrig, D. (2000). *Managing Software Requirements: A Unified Approach*. Reading, MA: Addison-Wesley.

Leif, V. (1998). Dynamic trade. May; www.forrester.com.

Lethbridge, T. C. (2000). What knowledge is important to a software professional? *IEEE Internet Computing*, May, 44–50.

McCarthy, J. C. (1999). The social impact of electronic commerce. *IEEE Communications, 37*(9), September, 53–57.

McConnell, S., & Tripp, L. (1999). Professional software engineering: Fact or fiction? *IEEE Software*, November–December, 13–18.

Meyer, B. (2001). Software engineering in the academy. *IEEE Computer*, May, 28–35.

OECD. (1996). The knowledge-based economy. Organization for economic co-operation and development, OCDE/GD(96)102, Paris, France.

Perkins, D. N. (1992). What constructivism demands of the learners? In T. M. Duffy & D. H. Jonassen (Eds.), *Constructivism and the Technology of Instruction: A Conversation* (pp. 161–165). Hillsdale, NJ: Lawrence Erlbaum Associates.

PIQ. (1998). Special issues on action learning, *Performance Improvement Quarterly, 11*(1–2).

Pour, G., Griss, M. L., & Lutz, M. (2000). The push to make software engineering respectable. *IEEE Internet Computing*, May, 35–43.

Repenning, A., Ioannidou, A., Payton, M. et al. (2001). Using components for rapid distributed software development. *IEEE Software, 18*(2), March/April, 38–45.

Revans, R. W. (1998). Sketches in action learning. *Performance Improvement Quarterly, 11*(1), 23–27.

Ryan, G. (1993). Student perceptions about self-directed learning in a professional course implementing problem-based learning. *Studies in Higher Education, 18*, 53–63.

Savery, J. R., & Duffy, T. M. (1995). Problem-based learning: An instructional model and its constructivist framework. *Educational Technology, 35*(5), 31–38.

Shaw, M. (1990). Prospects for an engineering discipline of software. *IEEE Software*, November, 15–24.

Speed, J. R. (1999). What do you mean I can't call myself a software engineer? *IEEE Software*, November–December, 45–50.

Stapleton, J. (1997). *DSDM: Dynamic Systems Development Method — The Method in Practice*. Reading, MA: Addison Wesley.

Umar, A. (1997). *Application (Re)Engineering: Building Web-Based Applications and Dealing with Legacies*. New York: Prentice Hall.

Wasserman, A. I. (1996). Toward a discipline of software engineering. *IEEE Software*, November, 23–31.

Wills, C. E. (1998). Group-based software engineering in an introductory computer science course. In *Proceedings of IEEE International Conference on Software Engineering: Education & Practice*, Dunedin, New Zealand, January 26–29.

Zachman, J. A. (1987). A framework for information systems architecture. *IBM Systems Journal, 26*(3), IBM Publication G321-5298.