Fakulti Teknologi Maklumat dan Sains Kuantitatif

Jilid 3, Bil. 1, Sept. 2001

JURNAL TEKNOLOGI MAKLUMAT DAN SAINS KUANTITATIF

KANDUNGAN **Muka Surat**

Universiti Teknologi MARA

To (Start With) OOP, Or Not OOP: That Is *Not* The Question*

Syed Ahmad Aljunid
Jabatan Sains Komputer
Fakulti Teknologi Maklumat dan Sains Kuantitatif
Universiti Teknologi MARA (UiTM)
40450 Shah Alam, Selangor, Malaysia

ABSTRACT

Lately, in UiTM, there has been a hue and cry against starting programming course with object-oriented programming (OOP), and opting to get back to the procedural paradigm first. This paper is an attempt to dismiss these notions. It argues that the wrong question has been asked. That problem is trivial. Instead, the more pressing question is: "How, in the present state of rapidly changing and expanding IT field, can we effectively teach fundamental programming principles and computer problem solving methodology without clouding the students with syntax and nontransparent tools?" Another related question is "How can we make programming attractive to the new students?" Essentially, it is not the question of when, but how. And what too. In short, we should actually revise our programming pedagogy thoroughly as well as review our programming curriculum. We present our case based on the successful implementation of the minimalist cum black box cum 4-stage approaches in introducing OOP to beginners of programming in various institutions. Our solution nevertheless takes into consideration our own background and constraints.

Keywords: Black-box approach; Minimalist approach; Object-oriented programming; Teaching of programming

Tel: +603 5543 5472
e-mail: aljunid@tmsk.itm.edu.my

* Adopted from William Shakespeare's Hamlet, Act III, Scene 1, line 67.

INTRODUCTION

UiTM has been teaching object-oriented programming (OOP) for its computer science majors since 1996. When the new curriculum was implemented in 1997, OOP was and is still being taught from the first semester onwards. Lately, there has been a hue and cry against starting programming course with OOP, and opting to get back to the procedural paradigm first. This paper is an attempt to dismiss these notions. It argues that even though these are genuine concerns, the wrong question has been asked. Instead, we propose to view the problem more holistically by restating it as two distinct but related questions: "*How*, in the present state of rapidly changing and expanding IT field, can we *effectively* teach fundamental programming principles and computer problem solving methodology in the first programming subject without clouding the students with syntax and nontransparent tools?" and "*How* can we make programming attractive to the *new* students?" Although our focus remains the first programming subject, we will approach the problem more holistically by viewing and reviewing the whole programming syllabus. Essentially, it is not the question of *when*, but *how*. In short, we should actually revise our programming pedagogy thoroughly. (Aljunid, 2000)

Nevertheless, we uphold the principle that changes in curriculum must be driven by data and facts rather than anecdotes, mere perceptions or intuitions. Thus we present our case based on the successful implementation of the minimalist cum black box cum 4-stages approaches in introducing OOP to beginners of programming in various institutions around the globe. Our solution nevertheless takes into consideration our own background and constraints.

We begin by viewing the current state of teaching of programming in UiTM in Section 2. Then, we will look at a few important considerations for reviewing the computing curriculum in general and the programming curriculum in particular in sections 3 and 4 respectively. In Section 5, we will discuss the teaching of programming in CS1 while in Section 6, we present several case studies of adopting OOP in introductory programming subjects using various ingenious pedagogical approaches before presenting our proposed solution in Section 7.

CURRENT STATE

Our faculty in UiTM has progressed over the years since starting the first computer science program in Malaysia in 1969. Thereafter, we have addressed the issue of teaching of programming periodically. In 1990, acknowledging the tendency among instructors to overdo with language constructs rather than the application of programming principles and methodologies, one of the resolutions of the 1990 UiTM Teaching of Programming Workshop (Aljunid, 1990) was "the teaching of programming must be focussed on application rather than syntax". The focal point is problem solving; 'Programming in X Language' rather than 'X Language Programming'.

However, over the years, this problem still persists, albeit at a lower rate, in introductory programming subjects. This problem was compounded when the new programming curriculum was implemented in 1997 for the Diploma in Computer Science (DCS) program (see Table 1 below), which is the feeder for our BSc (IT) program. Although we have spread out the OOP contents over three semesters using a single language, i.e. C++, the first subject, Fundamentals of Computer Problem Solving, continue to have problems. (ITC 120 contains, among others, computer problem solving, OOP concepts, objects, basic control constructs and classes)

Table 1: Current UiTM DCS/BSc (IT) programming curriculum

Code	Subject Name	Sem	Cr Hr	Category
ITC 120	Fundamentals Of Computer Problem Solving	DCS	14	Core
ITC 160	Object Oriented Programming I	DCS2	4	Core
ITC 210	Object Oriented Programming II	DCS3	4	Core
ITC 260	Data Structures	DCS4	4	Core
ITS 330	Information System Development	DCS5	4	Core
ITC 265	Commercial Programming	-	4	Elective
ITC 421	Programming Paradigms	BSc(IT)1	4	Core

Firstly, instructors differ as to when is it best to impart the rudimentary programming constructs associated with procedural programming such as control structures in this OOP-based subject. One camp prefers to start off with computer problem solving using these control constructs in the procedural paradigm and later introduce the OO implementation. The other camp prefers the computer problem solving to be taught from day one using the OO paradigm, and utilizing this control constructs set in the OO paradigm too. Of course, using C++ as the program language allowed such differences.

These instructors not only have different backgrounds, experiences and biases but also have different perceptions as to the receptiveness of their respective students in following their classes. Each has their own set of textbooks or references to back their particular approaches. Due to the fact that these roughly five hundred students for each intake sit for a common semester exam for all the six different UiTM campuses offering DCS in the country, uniformity of instruction has some degree of importance.

We argue that this problem is actually trivial. Although we personally would prefer the second approach, we opt to put the issue aside now*. After all, it makes no difference for the students in answering the exam paper. We further recognized that most instructors teach

* Refer to Section 5 (OOP in CS1) for a detail discussion on this issue.

it in the way they were taught. Incidentally, this 'approach' may not be pedagogically correct. For the time being however, instructor could and should continue teaching the first programming subject as to what constitute to him/her the best approach to deliver the materials until our proposal is accepted and the new curriculum is implemented.

Secondly, we noted, based on inputs from the instructors, the students are caught up with OOP nomenclature and concepts as well as C++ syntax such that they miss seeing the big picture. For example, defining and implementing C++ classes early in the semester peg down the students. The actual focus should be on computer problem solving and not on its tools per say. These nontransparent tools are presented to the new students too early such that they tend to block them for acknowledging the systems perspective.

What we need is gradual approach that will gently infuse the OO principles and methodology transparently in a systems perspective of problem solving so that they can interrelate between the objects or processes.

Thirdly, the students cannot appreciate the tiny, unrealistic problems that are given to them to be 'solved'. For example, building classes like *Circle* which only has methods like *setCircle(r), getRadius(), getArea()* and *getCircumference();* these unrealistic problems and text-based solutions do not attract them to programming nor show them the application of necessary skills for a well-designed computer solution.

We urgently need to overcome this problem by injecting real meaningful problems and GUI solutions as part of our attraction-grabbing maneuver (but without compromising core programming skills). In fact, the importance of objects and OOP will be driven home better via this approach.

In the end, we ask ourselves whether the students have developed the necessary and better problem solving skills as a result of exposure to the first programming subject.

CHANGING THE COMPUTING CURRICULUM

To design flexible computing programs that can keep pace with rapidly changing needs, Turner (1997) suggested among others, to (i) pay attention to pedagogy, (ii) shorten the time for adaptation of new curricula, and (iii) maintain stronger ties with the industry for faster feedback. He acknowledged that the role of pedagogy and the effective use of technology in computing programs have been neglected for the most part.

Taking into account the rapid changes in the IT field, the latest computing curricular guidelines from ACM/IEEE (Engel and Roberts, 2000), the call from the academia, the feedback from the IT industry as well as UiTM's current state and needs, we conclude there are at least three main issues that need to be addressed in designing the new computer science curriculum:

a) Inculcate systemic thinking and incorporate systems perspective of problem-solving
b) Emphasize skills in cooperative problem solving
c) Broaden the computing horizon

Inculcate Systemic Thinking And Incorporate Systems Perspective Of Problem-Solving

In the March 2000 draft of the Computing Curricula 2001 (CC 2001), The Joint Task Force on Computing Curricula noted that computing discipline has become much broader and had undergone various technical and cultural changes (Engel and Roberts, 2000). This rapid evolution of the computing discipline requires us to view with a systemic thinking into a lot of areas including problem solving.

The academia has also noticed this lack of systemic thinking phenomenon. Freeman (1997) from Georgia Tech noted that we can no longer view the components of the entreprise information system and their behaviors in isolation. He urged, "We must begin to take a systems perspective." Meanwhile, Stokes (1997) from Brigham Young University suggested to add a systems view of problem solving in calling for changes in our educational approach. The student must understand the dire need to integrate components into a comprehensive whole. They must be given the chance to develop complete systems as well as experience the consequences of incomplete designs.

With respect to computer problem solving and programming, Decker and Hirshfield (1993) succinctly described the present predicament: "It should not surprise us that our students can't appreciate the 'big picture' when what we spend most of our time teaching them in CS 1 is the 'little picture.'" Apart from the nontransparent teaching tools mentioned earlier, the choice of bottom-up enabled programming languages like Pascal and C also add to the crisis. These languages promote the tactical paradigm by focusing primarily on the details of coding and algorithms.

Feedbacks coming in from the industry also verified this problem. Haines (1997) from Boeing stated that one of the two outstanding reasons for project inadequacies and failures is the general lack of systemic thinking. Failing to grasp the whole picture of how processes interrelated made the projects doomed for failure. He further acknowledged the shortage of computing personnel with skills in systemic thinking, problem solving and communication.

Emphasize Skills In Cooperative Problem Solving

Stokes (1997) also suggested the above in calling for changes in our educational approach. Noting that learning to work with people is an essential skill for computing projects, he proposed small-group problem-solving exercises to help all participants understand the concepts better and promote teamwork.

Cooperative problem solving also emphasizes software reusability through the usage of components and class libraries. Software reusability drives home the "don't reinvent the wheel" mantra. In the same breath, teaching the value of using other people's well-developed code could increase the quality and stability of team-derived solutions.

Broaden The Computing Horizon

In the broadening of the computing horizon, the dramatic growth of computing and especially the pervasiveness of the WWW are crystal clear. The CC 2001 Task Force has not only acknowledged this emerging trend but has made it one of the central pillars for developing the new curricula. The number of knowledge areas has increased from nine in the CC'91 (Tucker et.al, 1991) to fourteen in CC 2001. Denning (1999), the current chair of the ACM Education and a member of the CC 2001 Task Force, has listed 24 different sub fields in the IT or computing or "informatics" profession.

Computing, which has strong historical roots in engineering, mathematics, and science, has finally emerged from these traditional ties. We, the computer scientists and engineers, must accept this reality and chart our new direction accordingly. We can no longer claim IT or computing to be our sole property. We can never separate computing from the emerging application areas and must work hand in hand with 'the news boys in the blocks'.

CHANGING THE PROGRAMMING CURRICULUM

In our opinion, the tools especially the programming language used in teaching of programming should at the very least enable and at best enforce a few guiding principles of programming. Spelt out by Kernighan and Plauger (1981) in their famous book, "Software Tools in Pascal", three of the principles are:

a) Keep it simple - clean, simple and clear code.
b) Controlling complexity is the essence of computer programming. Therefore, build it in stages, by concentrating first on the central and most important task.
c) Let someone else do the hard part — build on what you/others have done.

Although these principles were originally listed twenty years ago, they are still valid nowadays. We will dwell further on these principles in our solution in Section 7.

The IEEE/ACM CC'91 Task Force (Tucker et al, 1991) has made a number of recommendations on programming. We highlighted two of the more important points here. First, fluency in a programming language is an essential attribute of the mastery of programming. This implies there must be one dominant language in the curriculum. Secondly, teaching of programming must be in at least two other paradigms, such as functional, logic or OO besides the procedural paradigm. This knowledge will enhance both the student's appreciation and understanding of the various paradigms and tools currently available to solve a variety of problems in different domains.

In changing the computing curriculum and to answer our two initial questions, we openly acknowledge there exist more than one feasible path. At the same time, neither is we condemning structured programming nor gospeling OOP per say. We choose OOP because of three important reasons.

Firstly, OOP is the highly preferred mechanism in implementing systems view and inculcating systemic thinking right from the beginning. It enables us to look at a problem holistically and determine what software components would be required to produce a new computer-based solution. By abstracting entities into classes of objects, an organized overall view is obtained of the problem domain. By encapsulating implementation, the details have been well hidden when the problem is initially attacked. By enforcing message passing between objects, the importance of good and meaningful interfaces between inter-related elements in a system is driven through. By organizing the relationship between related classes of objects into a hierarchy format, the approach of designing programming solutions based on generalities is realized.

Secondly, OOP is the successful mechanism in applying cooperative problem solving in general and software reusability in particular. Initially coined in the Unix world as the "Don't reinvent the wheel" approach, software reusability was only realized with the advent of OOP. This is not just due to the inheritance property in OOP but more with the emergence of software components, design patterns and class libraries, which are all synonymous with OOP.

Thirdly, since OOP has been adopted, we preferred a feasible solution, which introduces changes gradually. In our eagerness to embrace new approaches and methods, we need not adopt the wholesale revamping of the existing system.

OOP IN CS1

The First Programming Language: Requirements And Objectives

Before we discuss the merits of OOP in CS1, let us first ask ourselves: is there a requirement for a first programming language? What are its objectives? There have been a substantial number of empirical researches on the most suitable type of language to teach programming concepts. Of course, the target novice group, whether secondary (high) school, college or university level does play a major role in deciding the language. In general, de Boulay et. al. (1989) concluded that a first programming language should have "syntactic simplicity with few special cases to remember". Walsh (1989) noted that syntactic difficulties might affect the student's ability to produce a logically sound, working program.

We formulated that among the objectives for a first programming language are:

a) It must be an effective medium in conveying the current fundamental programming principles, including those mentioned in Section 4, and computer problem solving methodology.
b) It allows students to see and solve problems systematically.
c) It enables students to make faster progress and deeper understanding of computer problem solving.
d) Its environment encourages and develops skills in cooperative problem solving.

Background On Teaching Of Programming In OOP For CS1

The common programming languages for teaching are essentially the same ones that we have used for the past 20 to 30 years. These include BASIC, FORTRAN, Pascal and C. Due to discontent with dominant, Pascal based curriculum taught in mid 1970's, several paths have surfaced. These include

- Functional programming – Scheme, LISP
- Procedural programming— C. (since too complex for novices, a subset of C is used), Modula 2, Turing.
- OOP -- C++, Java, Smalltalk, Eiffel
- GUI based: Visual Basic, Visual C++, C++ Builder

There are two different schools of thought on using OOP in CS1. One believes that structured programming must be taught first before OOP while the other believes in teaching OOP from day one. For example, Deitel belongs to the first school (1999) while Horstmann is a member of the second. Deitel and Deitel (1997).

The first school judges that jumping into objects at first is just too much for novices since new objects must be created first (Meter and Miller, 1994). The students must learn about objects, class specifications and definitions in order to program. Meanwhile, the second school considers moving from a procedural paradigm to an OO paradigm in one semester is too much for students to cope with. Instead, the approach taken is to gently introduce OOP in stages to students However, there is a growing consensus among the OO community to steep students in the OO paradigm before exposing them to the gritty details of hybird procedural/OO languages such as C++ (Meter and Miller, 1994). This is in tandem with our stand as stated earlier.

Generally, there is currently more acceptance of teaching OOP in CS1. It was even accepted more than five years ago. At the 1994 ACM SIGCSE Symposium on Computer Science Education, which discussed on integrating OOP concepts into the undergraduate curriculum, a large number of attendees advocated teaching OOP (in a variety of languages) in the first programming course for majors (Hirshfield et al, 1994).

The Challenge

A. The Instructors

In reality, the instructors themselves primarily inhibit the acceptance of OOP for CS1. Their reservations stemmed from misleading OOP myths. Decker and Hirshfield from Hamilton College tackled this issue wittily in a very interesting paper they presented at the ACM's SIGCSE Conference in 1994. Cynically, it was entitled "The Top 10 Reasons Why Object-Oriented Programming Can't Be Taught in CS 1" (Decker and Hirshfield, 1993), They noted, "Many of the reasons we came up with for not using OOP in CS 1 are seen to reflect our lack of understanding of the paradigm, our rear of the leanguate, and our past experience teaching Pascal and the procedural paradigm." It was the dreadful paradigm shift where all things fall back to zero that haunted them. Among the OOP myths that were dispelled by Decker and Hirschfield are " OOP is too hard for my CS 1 students!" (Reason #9), "The dreaded paradigm shift!" (#8) And "It screws up the rest of our curriculum!" (#3). The first and most important reason is of course "It's too hard for us!"

B. The Environment

The development environments used to teach OOP should include tools, which support OOP. However, Kolling and Rosenberg (1966) noted that, existing environments fail to fully support the object-oriented paradigm. They suffer from two deficiencies: insufficient object support and insufficient visualization support.

Object support allows objects to exist independently. This is the basis of the OO paradigm. It facilitates the possibility of incremental application development. However, the program/ procedure-oriented paradigm is still used in most environments. It is still used to build an application with exactly one entry point that can be compiled and executed only after all its parts have been completed.

Visualization support refers to structure visualization and manipulation techniques. Graphical visualization techniques should be used to display relationships between classes and objects while manipulation using the graphical or textual representation of a class should be possible interchangeably.

These deficiencies must be addressed to create an effective OOP development environment. However, these are beyond the scope of our paper.

CASE STUDIES OF TEACHING OOP USING VARIOUS PEDAGOGICAL APPROACHES

Case-Study 1: An American College Level
Approach: Black-box, minimalist, 4-stages
Language: Object Pascal
Target group: 1st year
Duration: one semester
Type: Actual Implementation
Rick Decker and Stuart Hirshfield from Hamilton College (Decker and Hirshfield, 1993)

A. The Problem

Hamilton College, like most other higher institutions, taught CS 1 using Pascal then. As widely known, Pascal and any other procedural languages impose an algorithmic method of problem solving. This enforces a tactical or bottom up approach, which focus on the details of coding. The principles of program design are not enforced by procedural languages.

As a result, the students generally have difficulty in analyzing and designing large-scale programs. Most could not produce reliable, verifiable, reusable and maintainable code.

This situation provoked a reaction: "Why are we investing an entire first course in teaching the equivalent of modern-day assembly language?" when the issues of systemic thinking, systems perspective of software analysis and design are missing?

B. The Solution: OOP in CS 1 using Object Pascal

Object-oriented languages, on the other hand, allow us to concentrate initially on top-level design considerations, and to progress — in true top down fashion — to working, workable programs that embody the principles of software engineering and program style.

Black box (Stehlik, 1993): Most of the program is hidden from the students, and the code is only exposed on an "as needed" basis. This approach not only mimics the process by which much software is developed but also allow the students to learn as apprentices from experts. (Meyer, 1993).

Minimalist: The details of the programming world, in particular, the algorithmic details of the language, are revealed slowly and only as needed to illustrate the principles of objects. These include algorithmic constructs like assignments, selections, and loops.

4-stages: Four stages are employed. (i) Initially, students *use* the object world by interacting with sample programs that are provided to illustrate specific OOP and Object Pascal features. (ii) As they become more familiar with objects, students are exposed to the details of the world so that they can *read* the implementations of the objects that they have been using. (iii) Then, they are shown how they can *modify* and *extend* the existing object world to suit the needs of a particular program. (iv)Finally, they are taught how to *define* their own objects.

Central to each set of lab exercises is a full-scale, working, sample program.

Results: With appropriate pedagogy, OOP can be taught from the start. This approach leads students naturally to appreciate the virtues of OOP in their first programming experiences by allowing—even encouraging—them to work like professional programmers from the start of their programming careers.

Case-Study 2: A Top American University Level
Approach: Literate, situated, black-box, minimalist, case-study based
Language: Karel the Robot, Pascal, Object Pascal
Target group: non-CS majors
Duration: one semester
Type: Experiential

Glem Meter and Philip Miller from the School of Computer Science, Carnegie Mellon University (Meter and Miller, 1994)

Literate programming: visual development environment (Pascal Genie) with multiple views and visual debugger.

Situated programming: teaching (OO) programming using an example thoroughly embedded in a domain other than computer science

Case study based: Students learn by working in programs developed by experts.

OOP in CS1 implementation: Start with procedural programming with Karel the Robot for 2 weeks, then with Pascal (up to arrays of records and searching), while the final third of the semester is done using Object Pascal and artificial life (as the application domain) using an example thoroughly embedded in a totally non-computer science domain.

Results: The students were intellectually engaged. They came to grips with basic object-oriented programming, mastered the topics of procedural programming, learned first hand about computer simulation and learned a bit about the applied domain. Most importantly, through it all they used programming to express and explore their own powerful and novel ideas.

Case-Study 3: An Australian University Level
Approach: Incremental, black-box
Language: Smalltalk
Target group: secondary school students
Duration: six hours
Type: Experiential

Andrew Hussey, David Leadbetter and Helen Purchase, from the School of Information Technology, The University of Queensland (Hussey et al, 1996)

Objective: Teaching object-oriented programming skills in six hours to school students whom have either never programmed or never encountered OOP before.

Goal: To devise an educational strategy that would ensure that at the end of the six hours, the students would have grasped the basic concepts, to the extent that they could make non-trivial changes to existing programs, and show understanding of the concepts through discussion.

Implementation: Smalltalk using the VisualWorks environment

Strategy: Use many small, guided exercises of graduated difficulty. Small independent exercises reinforced material soon after it was introduced. Provided students with a black-box view of an example system, which, was progressively opened as their knowledge grew. The overall strategy for each project (puzzle and calculator projects) was to introduce the application being used, first from a user's perspective, and then from a programmer's perspective.

Results: Majority of the students understood the basic concepts of programming and OOP to the extent that they could discuss and use them. They enjoyed the sessions and exhibited enthusiasm for the language and the Visual Works environment.

SOLUTION

Overall Solution

Our proposed solution is based on the extensive arguments, views, experiences, recommendations and case studies presented above. As noted earlier, the issue is not whether to start with OOP or procedural but rather to actually revise our programming pedagogy thoroughly as well as review our programming curriculum. We present our case based on the successful implementation of the minimalist cum black box cum 4-stages approaches in introducing OOP to beginners of programming. Our solution nevertheless takes into consideration our own background and constraints.

Referring to the three case studies given above, we purposely picked three different levels, target group, duration and type of case studies. Case I has the most similar background, target group and level compared to UiTM, and was an actual implementation whereas cases II and III were both experiential in nature. Case II involved a set of highly talented but non-CS majors while Case III involved a set of secondary school leavers. Case II shows that a thoroughly pedagogical approach equipped with the necessary environments and expertise can yield successful and intellectually engaged results while Case III demonstrated it is possible to teach OOP to those without programming background in a very limited time frame *if* the right pedagogy is utilized.

Our proposal is based on the following emphasis:

1. Inculcate systemic thinking and incorporate systems perspective in problem solving.
2. Instill skills in cooperative problem solving via teamwork and by utilizing components and classes library.
3. Keep the solution simple.
4. Control complexity by building it in stages.

Minimalist C++ Language

In the first programming subject (ITC 120), our approach is to gently introduce important general and OO programming concepts via a GUI OOP package such as Borland's C++ Builder® or Microsoft Visual Basic® before proceeding to a 3GL OOP language (C++). By using class libraries and components from the onset, we try to encourage the students all these virtues that have been singled out above.

The core of our pedagogical approach is realized via the minimalist cum black box cum 4-stages approaches. This set of approaches will be similarly implemented for all the other programming subjects although we can relax the 4-stages approach where appropriate.

Rather than changing to a simpler OOP language such as Eiffel or to a pure OOP language like Java, we suggest to stick to C++. Even then, we continue to highlight certain mechanisms while de-emphasizing C++ language peculiarities such as pre and post increments (++j vs j++) and operator overloading. This minimalist language approach sends the message that the language is secondary: paramount are the ideas and secondary is the computational representation of the ideas (Meter and Miller, 1994). Our decision to use C++ is primarily to lessen the changes and make these changes gradual, thus encouraging more acceptances from the instructors. As Cassel (1997) has aptly puts it, "A truly effective solution carefully considers the potential value of old methods as well as embracing the obvious benefits of new ones."

Assignments and Projects

We noted that most programming assignments given throughout the whole programming curriculum are small in nature and cover a very specific topic or sub-topic of programming. Only a few assignments are varied and applied domain problems. We suggest that for every programming subject, there must be at least one project (but preferably two so as to allow them to learn from their mistakes; the first project is a short one) that requires them to utilize most of the principles, methodologies and mechanisms learned in that subject We recommend to add more applied domain problems too.

On programming projects, we noted only the ITC 421 currently has one. We propose to start team-based projects from ITC 210 onwards. Of course, all these projects must use components or class libraries, either from standard libraries or from libraries that have been developed by their instructors or seniors.

Specific Programming Languages, Data Structures and Software Design Patterns

We noted that those graduating from the DCS programs do not have the knowledge of a few different programming paradigms and languages. Similarly, those taking the BSc (IT) program do not possess the necessary programming knowledge and skills required by the industry as the only core-programming subject, ITC 421, "Programming Paradigms", is more of a survey of programming paradigms and languages. Specifically, we observed that these BSc (IT) graduates do not possess knowledge of some basic data structures like B-trees, inverted files and graphs. Likewise, they do not know software design patterns, which since put forth by Gamma et. al. (1995) has been an important part of the software engineering field.

All those points mentioned above have been verified in an extensive survey conducted by Lethbridge (1998) from University of Ottawa in 1998. The majority of the software professionals indicated that specific programming languages is both the most important and most learned topic among the 75 topics selected. Second is a data structure third are software design and patterns while object-oriented concepts are six.

Thus apart from the changes proposed for ITC 120, the following are other changes:

- ITC 260 is renamed ITC 26x*, "Data Structures I". We may have to move one topic up to the new ITC 4xx*.
- A new subject, ITC 200*, "Survey of Programming Languages" (similar to the old CSC 117) at DCS level, and can be taken after completing ITC 160. It must cover at least two different paradigms. One is procedural; the other can be chosen among functional, logical, Web-based, etc. It should not contain Java or any OOP based language.
- ITC 4xx*, "Data Structures II and Software Design Patterns" as replacement for ITC 421, "Programming Paradigms" (PP). ITC 4xx will cover "advanced" data structures like B-trees, threads, inverted files, graphs and also software design and patterns. A medium size (at least 500 lines of code) team-based project should be a requirement. ITC 4xx is not just for BSc (CS) but also BSc (IT). Currently, the ITC 421 subject is too shallow since it is the only required programming subject at bachelor level.

Lastly, we propose the new programming curriculum to implement all subjects in three credit hours. We should do this not just for the sake of standardizing with most universities but more importantly, an actual three credit hour for a core subject is actually sufficient. We can either implement it in a three-hour lecture format, or a two and a half followed by a one-hour lab format. The following is the proposed curriculum

* the specific code will be determined later

Table 2: Proposed UiTM DCS/BSc (IT) programming curriculum

Code	Subject Name	Sem	Cr Hr	Category
ITC 120	Fundamentals Of Computer Problem Solving	DCS1	3	Core
ITC 160	Object Oriented Programming I	DCS2	3	Core
ITC 210	Object Oriented Programming II	DCS3	3	Core
ITC 26x	Data Structures I	DCS4	3	Core
ITS 330	Information System Development	DCS5	3	Core
ITC 265	Commercial Programming	-	3	Elective
ITC 2xx	Survey of Programming Paradigms	DCS 3-5	3	Core
ITC 4xx	Data Structures II and Software Design Patterns	BSc (IT)	3	Core

CONCLUSION

We have proposed a new programming curriculum based on the current needs and rapidly changing computing field. Our approach is rich in pedagogy, which has been neglected for a long time. We also try to capture the student interest in programming as well as being practical in our approach. We try to realize all this via the minimalist cum black box cum 4-stages approaches in introducing OOP.

Acknowledging that changes in curriculum must be driven by data and facts rather than anecdotes, mere perceptions and intuitions, we have managed to substantiate our proposal based on extensive arguments, views, experiences, recommendations and case studies taken from reputable and varied sources.

We must have the strength and courage to change and adapt accordingly, and to act fast too. As Haines (1997) puts it, "The pace of computing technology evolution demands that industry and academic institutions work more closely together. ... Perhaps the academic community needs to "reengineer" itself in much the same way as most of the industry has done or is doing. There may be no other way to equip college graduates to meet industry expectations."

We end our paper by reflecting one of the resolutions of the 1991 UiTM Programming Workshop Resolutions (Aljunid, 1991): "We must instill an academic culture which is open to criticism, able to accept changes including changes in direction as well as to embrace any new paradigm shifts".

REFERENCES

Aljunid, S.A. 2000. The Teaching of Object-oriented Programming – where do we stand? *Bengkel Pengajaran Pengaturcaraan UiTM 2000 (BPPU 2000),*
UiTM, Shah Alam, Jun 8.

Aljunid, S.A. 1990. *Bengkel Pengajaran Pengaturcaraan*, Kajian Sains Matematik dan Komputer, UiTM, Shah Alam, December.

Boulay, B.D, O'Shea, T, Monk, J. 1989. The Black Box within the Glass Box: Presenting Computing Concepts to Novices, in *Studying the Novice Programmer*, E Soloway and J Spohrer (ed.), Lawrence Erlbaum Associates Inc, NJ.

Cassel, L. 1997. Blend of Old and New. *IEEE Computer*, 30(11): 50-51, Nov.

Decker, R and Hirshfield, S. 1994. The top 10 reasons why object-oriented programming can't be taught in CS1. *Selected papers of the twenty-fifth annual SIGCSE symposium on Computer science education*, 51-55.

Decker, R and Hirshfield, S. 1993. Top-down teaching: object-oriented programming in CS 1. *Proceedings of the Twenty-fourth SIGCSE technical symposium on Computer science education*, 270 – 273.

Deitel H. and Deitel. P. 1997. *C++ - How to Program*, (2nd ed.), Prentice-Hall, New Jersey.

Denning, P.J. 1999. Our Seed Corn is Growing in the Commons. *Information Impacts Magazine*, March.

Engel, G and Roberts, E. (ed.) 2000. Computing *Curricula 2001*, The Joint Task Force on Computing Curricula, Association for Computing Machinery and the IEEE Computer Society, (draft), March 6.

Freeman, P.E. 1997. Elements of Effective Computer Science. *IEEE Computer*,
30(11): 47-48.

Gamma, E, Helm, R, Johnson, R and Vlissides, J. 1995. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading.
Haines J.E. 1997. The Case for More Relevant Computing Skills. *IEEE Computer*, 30(11): 55-56, Nov.

Hirshfield, S., Astrachan, O., Barr, J., Donnelly, K., Levine, K., and McGinn, M. 1994.

Object-oriented programming: how to scale up CS 1. *Selected papers of the twenty-fifth annual SIGCSE symposium on Computer science education*, 396.

Horstmann, C. 1999. *Computing Concepts with C++ Essentials* (2nd ed.), John Wiley, New York.

Hussey, A, Leadbetter, D and Purchase, H. 1996. Learning object-oriented programming in six hours: an experience with school students. *Proceedings of the second Australasian conference on Computer science education*, 117 – 125

Kernighan, B. and Plauger. 1981. *Software Tools in Pascal*, Addison-Welsey.

Kölling, M and Rosenberg, J. 1996. An object-oriented program development environment for the first programming course. *Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education*. 83 – 87.

Lethbridge, T. 2000. What Knowledge Is Important to a Software Professional? *IEEE Computer*, 33(5): 44-50, May.

Meter, G and Miller, P. 1994. Engaging students and teaching modern concepts: literate, situated, object-oriented programming. *Selected papers of the twenty-fifth annual SIGCSE symposium on Computer science education*, 329 – 333.

Meyer, B. 1993. Toward an Object-Oriented Curriculum. *Journal of Object-Oriented Programming*, 6(6): 76, May.

Stehlik, M. 1993. Report for the panel *Approaches to Programming Assignments in CS1 and CS2*, Proceedings of the Twenty-Fourth SIGCSE Technical Symposium on Computer Science Education, March.

Stokes, G.E. 1997. Rethinking the Current Formula. *IEEE Computer*, 30(11): 48-49, Nov.
Tucker, A.B, Barnes, B.H, Aiken, R.M, Barker, K, Bruce Kim B, Cain, J.Thomas,. Conry, S.E, Engel, G.L, Epstein, R.G, Lidtke, D.K, Mulder, M.C, Rogers, J.B, Spafford, E.H, and Turner, A.J. 1991. *Computing Curricula '91*, Association for Computing Machinery and the IEEE Computer Society.

Turner, J. 1997. Making the Structure More Flexible. *IEEE Computer*, 30(11): 56-57, Nov.

Walsh, T. 1989. Rationale for and Problems Involved in Teaching Programming. *M.Ed. Dissertation*. University of Stirling, Stirling.