DevShops : Bridging the gap between Academia and the real world

Sandra Cleland
UCOL
s.cleland@ucol.ac.nz

ABSTRACT

This multi-year, action research study evaluated the effectiveness of a real-life, team based software development project at preparing software engineering students for the workplace. Graduates, working in software development roles, were surveyed to determine how useful the experience had been, and to find common gaps in their skill sets. The learning environment was adapted to address the identified skill gaps and to keep abreast with technology changes.

Keywords: work experience, industry relevant skills, team-based learning

1. INTRODUCTION

It has long been recognised that traditional software engineering education has been lacking in teaching students skills that will make them ready for the challenges of the workplace. Research suggests that graduate developers are ill-prepared for the realities of their first position (Begel & Simon, 2008a, 2008b; Borenstein, 1992; Brechner, 2003; Dawson, 2000; Dawson, Newsham, & Fernley, 1997). Traditionally, most software development courses involve project work where students develop a software application from scratch based on stable user requirements. This does little to prepare them for the type of software maintenance work they are likely to be given as graduates entering the workforce. To better prepare students for transition into the workforce it is generally agreed that software development students should: work as a team using team-based development tools, be exposed to an application with a large existing code base, have a real-life client or role play of such, have a project with changing requirements or be made to prioritise conflicting requirements. UCOL's Bachelor of Information and Communications Technology (BICT) students undertaking their 300 level Software Engineering course are involved in a real-life project where they have to work as a team in order to better prepare them for the real world. This multi-year action research study involved: surveying of graduates of the course to determine the extent of skills developed within this learning environment, to identify how useful the skills have been, and to find common gaps in their skill sets experienced as new employees; adaptation of the learning environment to address the identified skill gaps; observation of students working within the adapted environment, and analysis of their reflective journals. This paper begins with an historical overview of Software Developer education and then Industry perspectives on the education and skill requirements of junior developers are outlined. The background literature review is followed by a description of the study including: the learning environment and the project task, graduate feedback on skills developed, and adaptions that were made based on the feedback.

2. A BRIEF HISTORY OF SOFTWARE DEVELOPER EDUCATION

Bauer (1973) discusses the evolution of programming and programming education. He describes three stages of progression in the early decades of the computing era: the classical period; the pioneering age; and the software crisis (Bauer, 1973). The classical period, from 1945 – 1955, saw the arrival of the first fully mechanical, programmable computer and was dominated by the engineers who built these early machines; therefore the fascination was in the technical capabilities of the machine (particularly the electronics) and not the programming of it (Bauer, 1973). This led to neglect of the software side of things and resulted in no training of programmers in this period; programming was something the specialist (e.g., engineer, physicist, mathematician) had to do for themselves (Bauer, 1973).

The next period, the pioneering age from 1955 -1965, saw the commercial use of computers begin which resulted in an explosion of sales of the machinery and an increasing "demand for personnel to be delivered with the computer", refer Figure 1 (Bauer, 1973, p. 472).

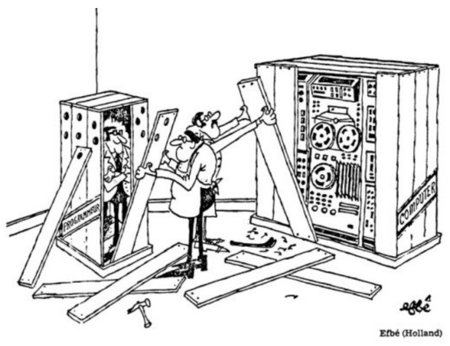

Figure 1: Programmer in a box

The manufacturers coped with the demand by cajoling as many intelligent beings as they could into the field, leading to programmers with little or no training and "an unwillingness to catch up" (Bauer, 1973). The need for formal training was being felt and many universities responded with Computing Science or Computer Science, the choice of term was under dispute as was the content of the courses (Bauer, 1973). Bauer (1973) states that during this time some of the problems faced by educational institutes were to do with establishing a Ph.D in the field of Computer Science; this also influenced peripheral subject choices as did the location of the department within the university itself. For example: the areas of automata theory and artificial intelligence were frequently enhanced under the Computer Science banner as it was both easier to obtain a professor and gain a Ph.D in these fields than in system programming (Bauer,1973). Bauer (1973) quotes Perlis to sum up the state of programming education at this point: "What is good for computing science is what is good for writing a Ph.D thesis in computing science" (p. 474). So despite the attempts of educational institutes to start to

This quality assured paper appeared at ITX 2014, incorporating the 5th annual conference of Computing and Information Technology Research and Education New Zealand (CITRENZ2014) and the 27th Annual Conference of the National Advisory Committee on Computing Qualifications, Auckland, New Zealand, October 8-10, 2014. Mike Lopez and Michael Verhaart, (Eds).

address the need for programming training the pioneering age concluded with still very few experienced and able system programmers (Bauer, 1973)

In the next decade, the computing era entered into the period known as the "Software Crisis" which was characterised by computer systems that were being built bigger and bigger, as there were no costly raw materials, but definitely not better (Bauer, 1973). The increasingly large systems consumed large amounts of expensive hardware storage and suffered performance problems, they were also unreliable as they were full of flaws (Bauer, 1973). Hamming (cited in Bauer, 1973) summarised the issues of the software crisis: systems were not delivered on time and they often didn't run; systems that did run were full of bugs; programmers were poor communicators who were unable to deliver on time and co-operate with others. Bauer (1973) points out that during this period the way in which manufacturers set up maintenance arrangements for their software could only lead to the conclusion that the users paying for the product were expected to field test the software and find the bugs. Manufacturers would only fix the bugs when the client demanded help and often the fix would introduce new bugs, which would again be ignored (Bauer, 1973). This type of behaviour led Dijkstra (cited in Bauer, 1973), a participant at the NATO software engineering conference in 1968, to state "the whole business is based on one big fraud" (p. 476). During the software crisis period there were major improvements in the techniques used for software development, such as the use of control structures and decreased use of the goto statement as advocated by Dijkstra; techniques that came to be known as structured programming (Dijkstra, 1968). These new practices started to flow through to some of the Computer Science curriculums. Bauer (1973) closes his summary of programming history with a look at the educational problem faced in the decade that would follow, 1975-85. He expresses the need for "fundamental changes in the habits of the programming community" that can only be bought about by education; he suggests that there should be a supervised practical work component to the education of software developers so they can practice their skills while being mentored (p. 477). He concludes by stating that future graduates need to make software development a profession by using engineering practices with a scientific viewpoint (Bauer, 1973).

The computing industry was growing rapidly and educators were struggling to keep pace with the changes in this new field. The computer science courses were being criticised as not meeting the needs of industry. Borenstein (1992) states that the skills focused in on by these courses are unrelated to the tasks graduate programmers will be engaged in once employed.

In 1976, the first suggested framework for a separate software engineering curriculum was proposed by Freeman, Wasserman, and Fairley (1976). Freeman et al. (1976) describe the software engineer as a generalist who takes on a multitude of roles: "problem solver, designer, implementor [sic], manager, facilitator, and communicator" (p. 116). To provide the skills needed by these generalists who worked in projects where there is often time and budget constraints, as well as unclear user requirements, five content areas were suggested to underpin any future training programs in software engineering; the areas to be covered were: computer science, management science, communication skills, problem solving, and design (Freeman et al., 1976).

Freeman et al. (1976) provided an assessment of the state of software engineering education at the time the framework was proposed. They describe the beginning of the introduction of software engineering courses into the graduate level offerings of the computer science departments; commonly these courses tried to combine delivering theory principles (via lectures, discussions and readings), with application of the principles through a project (Freeman et al., 1976). These projects are often conducted in small groups and the emphasis is on the method used throughout the project rather than the end product (Freeman et al., 1976). Ten years on Freeman (1987) wrote of the failure of most educational institutions, and the computing industry in general, to progress software engineering practices.

In 1988, a curriculum design workshop was held by the Software Engineering Institute at Carnegie Mellon University to design a model for a Masters of Software Engineering which was to be a "terminal professional degree", designed for the practitioner with a project or practicum element being used to demonstrate knowledge attained (Ardis & Ford, 1989, p. 8). The result of the workshop was a curriculum made up of six core courses, a project component, and electives making up 20 – 40% of the program. The experience component was suggested to be at least 30% of the student's work and could be offered in a variety of ways such as: a capstone project - where student completes a project after completion of a majority of coursework; a continuing project – where students are part of an in-house software factory that works on an on-going application development task; a multiple course coordinated project – where "a single project is carried through four courses (on software analysis, design, testing, and maintenance)"; an industry cooperative program – where students go out to industry for six months then return to complete their studies; a commercial software company – where students work in a commercial operation established by the university in cooperation with local industry; a design studio – where students work on a project in an apprenticeship type relationship under the guidance of an experienced developer (Ardis & Ford, 1989, p. 42).

The examples above of how to provide experience to software engineering students were all currently in use at the 15 universities, who at the time, offered the only graduate software engineering programs (Ardis & Ford, 1989). The two most common approaches being the capstone project and the inclusion of a project in a lecture course (Ardis & Ford, 1989). One of the earliest reflections by researchers on the course integrated project experience comes from the University of Toronto's "Software Hut" first delivered in 1973 (Horning & Wortman, 1977, p. 325). Each software hut was a team of three tasked with design and creation of one of two software components; following this teams were required to purchase a module from other software huts and interface them with their own; a later phase involved modifying a system consisting of the two components built by other software huts and adding to the program specifications so changes to the software were required (Horning & Wortman, 1977).

Though there now had been good progress made to define what a graduate level course for software engineers should comprise, educational institutes were still sluggish to address the need for software engineering practices to be taught throughout undergraduate level programs. A 1994 report from the Software Engineering Institute (SEI) states the researcher found no "undergraduate programs named bachelor of science in software engineering at any United States universities" (Ford, 1994, p. 3). Ford (1994) goes on to describe the efforts of eleven universities to address the need for an undergraduate software engineering curriculum. These institutes offered either a sequence of software engineering courses in their undergraduate computer science program or an undergraduate program that is software related (e.g. Bachelor of Science in Software Engineering Technology) (Ford, 1994). The former approach of a sequence of software engineering courses within an undergraduate computer science degree was the most prolific, with Ford (1994) stating that nearly all computer science programs now included at least one software engineering course.

So, it had become common practice to include one or more software engineering courses into existing undergraduate Computer Science programs to try to satisfy industry demands for skilled practitioners (Ford, 1994; Leventhal & Mynatt, 1987). Ford (1994) speculates that the slow growth of the dedicated software engineering curriculums was due in part to the well-established computer science degrees providing programming graduates that were proficient enough for an inefficient industry. As the size and complexity of the systems being developed grew, so would the demand for skilled software engineers; the differences between the disciplines would become more apparent and the need for separate educational programs more urgent (Ford, 1994).

To find out what a typical undergraduate software engineering course was comprised of, Leventhal and Mynatt (1987) surveyed a sample of USA and Canadian institutes offering a Bachelor's degree in Computer Science. They state that "the typical course focuses on the software development life cycle and involves a significant project worked on by teams of students, with student leaders" (Leventhal & Mynatt, 1987, p. 1197). Early life-cycle courses cover topics in the early stages of the software development life cycle: requirements analysis, requirements specification, and system design (Leventhal & Mynatt, 1987). A large part of the student grade is based on a project and these courses make heavy use of written reports to document project outcomes, reflecting the actual deliverables of these phases in industry (Leventhal & Mynatt, 1987).

Later life-cycle courses were the most predominant of the course types and cover the phases of the software development life-cycle that produce functioning code: detailed design, coding, testing, and maintenance (Leventhal & Mynatt, 1987). Leventhal and Mynatt (1987) state this type of course is product oriented with a large portion of the students grade related to a project. The later life cycle phases, where code is produced, have the most in-depth coverage of all of the topic areas in the software engineering courses and are perceived by faculty as being more effective than those focusing on the non-code producing elements of software engineering practice (Leventhal & Mynatt, 1987).

Over 95% of the institutes surveyed included a project component in at least one of their software engineering courses; a feature that was common across all three course types (Leventhal & Mynatt, 1987). Projects were combinations of toy projects and ones intended for actual use, with the lecturer role playing the user; a majority of institutes made use of teams at least some of the time, with 90% reporting that teams were used frequently (Leventhal & Mynatt, 1987). Most courses have the student teams work on different projects with only a small percentage engaging the whole class in one project; a majority of the project teams are student led and the project accounts for 40% or more of the student grade, with the lecturer normally having the entire responsibility for grading of the project (Leventhal & Mynatt, 1987).

Shaw and Tomayko (1991) describe five different models of project work used in one semester, undergraduate software engineering courses. All of the models assume students have knowledge on program construction from prerequisite courses, with the aim of the software engineering course being "to demonstrate how this technical material is applied in the context of large scale software development" (Shaw & Tomayko, 1991, p. 38). The five models (refer Figure 2) are: software engineering as artefact, topical approach, small group project, large project team, and project only (Shaw & Tomayko, 1991, p. 40).

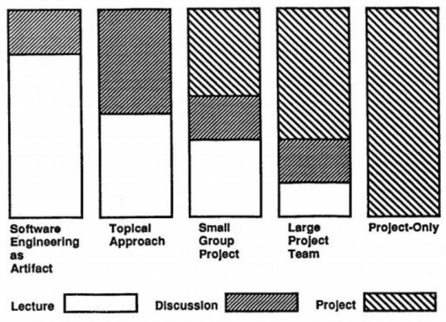

Figure 2: Models of the One-Semester course

Described as the "all-talk, no-action" models, the software engineering as artifact model and the topical approach model rely heavily on lectures and some discussion to teach software engineering concepts (Shaw & Tomayko, 1991, p. 39). These two approaches have the advantages of: being able to fit nicely into an academic semester or quarter, and allowing the students to focus on issues presented by the instructor rather than constantly troubleshooting project problems (Shaw & Tomayko, 1991). However, Shaw and Tomayko (1991) point out that two key areas that lead to project failure, communication issues and configuration control, cannot be appreciated until students experience working with others on a software product.

The small group project, the most common model in use at the time, splits the course evenly into project work and class work (Shaw & Tomayko, 1991). The project is limited in scope and size due to the constraints of it needing to be completed within a term by a group of between three to five students (Shaw & Tomayko, 1991). Shaw and Tomayko (1991) describe this small team, small project, fake customer approach as really just an extension of "the programming-in-the-small" experience students are likely to have had during their computer science course work but they indicate that by paying particular attention to configuration management and quality assurance larger project issues can be taught (p. 39).

Shaw and Tomayko (1991) assert that if the course "is to immerse the student in a practical, real-life, software product development process" then the large project team model is the way to do it (p. 40). This approach has the students all take on different roles (e.g., designers, quality auditors, configuration managers) within one large project team to work on one software product; there is often a real customer and students learn about the other roles within the team through the normal interactions that happen during the project (Shaw & Tomayko, 1991). The project only model is similar but rather than spend any course time discussing software engineering topics, students learn entirely by being immersed in the project; this is a common approach of the capstone course (Shaw & Tomayko, 1991). At this point in the history of Software Developer education Industry was striving to establish Software Engineering as a Profession, this also involved an Industry driven effort to define a model for a Software Engineering undergraduate curriculum.

3. AN INDUSTRY PERSPECTIVE

In 1993, the Association for Computing Machinery (ACM) and the IEEE Computer Society created a joint Steering Committee for the Establishment of Software Engineering as a Profession, five years later this became the Software Engineering Coordinating Committee (SWECC) (Duggins & Thomas, 2002). One of the main projects of this committee was the Software Engineering Education Project which established guidelines for software engineering curriculums; these guidelines were formally defined by the joint ACM/IEEE committee in a report known as SE2004 (Lethbridge, LeBlanc, Sobel, Hilburn, & Diaz-Herrera, 2006). SE2004 identifies ten knowledge areas that are fundamental to the education of software engineers, these are commonly referred to as SEEK (software engineering education knowledge) (Lethbridge et al., 2006).

SEEK is outcome focused and identifies specific tasks software engineering graduates should be able to perform and behaviours they should possess. These included: being able to work in teams as well as individually; being able to make good decisions around system compromises necessary due to the common project constraints of time, cost, and existing systems; being ready for work due to mastery of software engineering knowledge and skills; being able to make ethical and economically sound design decisions in one or more domains; being able to apply current techniques and models appropriate for the phase of the systems development life cycle; having a commitment to lifelong learning necessary for the rapidly changing environment; being able to communicate; and being able to negotiate effectively as well as demonstrating effective work habits (Lethbridge et al., 2006).

Hilburn (1997) reports that graduates are not lacking in technical skills, they are lacking in people and process skills. As new employees they: struggle to communicate effectively, have little experience in team work, are unable to effectively manage their workload, lack appreciation of organisational structures, and have little understanding of business processes (Hilburn, 1997). A recent investigation conducted by the Australian Learning and Teaching Council reports that employers find it can take anywhere from three to 12 months for “graduates to get up to speed in industry”, often costing companies more than $10,000 per graduate (Koppi & Naghdy, 2009, p. 8).

In 1983, a company called Plessey Telecommunications (later renamed GPT and now known as Siemens GEC Communication Systems Ltd) started an in-house training course for their computer science graduates with the aim of highlighting “the differences of working in the real world compared with the near ‘ideal’ environment experienced at university” (Dawson et al., 1997, p. 287). The course was instigated by the companies software managers who were finding that new graduates took up to six months to become productive; the productivity delays were not only to do with the graduates’ learning curve around the company’s tools and product lines, it also involved their lack of readiness to deal with the realities of the workplace (Dawson, 2000). New employees participate in a two week group project as part of a team of four or five; the first day involves a set up session where the project aims are outlined and graduates are made aware of difficulties that are often faced in real software engineering projects; the final day includes a review of how the project went with the remaining time being devoted to the development task (Dawson, 2000).

During the two week project the course instructor role plays the customer, manager, and other key personnel, such as a quality auditor, and a number of what Dawson (2000) refers to as “dirty tricks” are played on the teams (p. 210). Dawson (2000) describes these dirty tricks as being the defining difference between the real-life project experiences that graduates may have had during their education and the real-life experience they are given at the in-house training course. He describes the educational approach of trying to provide the best possible environment for students to learn in as being detrimental, with students having an unrealistic view of what is involved in the real world of software development (Dawson, 2000).

Dawson (2000) outlines the 20 dirty tricks that have been used in various combinations during the in-house training course to simulate real work conditions: give them a vague initial specification with ambiguous statements and missing detail; make all their assumptions incorrect, in particular reject any additional features they have added; role play a customer who does not know what they want and has low computer literacy; change the requirements, or re-prioritise them, regularly; have conflicting requirements; provide customers with conflicting ideas; have customers with different personalities (e.g., super enthusiastic vs. super reluctant); ban overtime; add tasks so the schedule is disrupted; move deadlines; have quality auditors schedule inspections randomly; change the truth slightly and then deny that there has been any change; swap team members around; change the work practices to simulate changes in management; upgrade software mid-project; change the hardware platform that the software will run on towards the end of the project; crash hardware; slow down the development software by overloading the network so builds and testing take twice as long; delete files so teams have to revert to the last backup; finally, say I told you so when the graduate developers express their frustration.

Of these 20 dirty tricks about half are used during a course with only the inadequate specification and regularly changing requirements always being included (Dawson, 2000). Over the time this training course has been run the company has seen a steady improvement in the work readiness of graduates with more institutes providing real-life project experiences; however, they believe there is still significant work to be done before the in-house training course becomes redundant (Dawson et al., 1997).

In 2008, an observational study undertaken by Begel and Simon (2008b) on new graduates in software developer roles at Microsoft discussed that while the graduates were proficient in: coding, reading and writing of design specifications, and problem solving; they experienced difficulties in the areas of: communication, collaboration, technical tools used for large-scale development, cognition, and orientation within their project teams where there may be little in the way of well organised information. Begel and Simon (2008a) found communication was the highest frequency activity for the novice software developers, and was deemed to be critical to productivity. The novices communicated to: coordinate activities, ask for help, participate in meetings, report to managers, work with others, get feedback, give help, find people, and persuade others (Begel & Simon, 2008a). Novice software developers encounter difficulty over knowing when and of whom to ask questions, often showing reluctance to ask for help early so as not to appear incompetent (Begel & Simon, 2008b). Collaboration within large teams and working alongside multiple teams were major areas of difficulty and uncertainty for the novices; some found themselves being given tasks by co-workers which were not actually set by the manager but still undertook them without complaint (Begel & Simon, 2008b).

Begel and Simon (2008b) summarise the major difficulties experienced by the novice software developers under the following categories: communication, collaboration, technical, cognition, and orientation. The novice developers suffered communication issues around: knowing when and how to ask questions; asking questions with the appropriate amount of information for the circumstance (Begel & Simon, 2008b). Novice developers who were not native English speakers struggled with abbreviations being pronounced as words, and with words with different meanings but the same pronunciation, often leading to miscommunication (Begel & Simon, 2008b). Collaboration problems were around: adapting to working with a large team, interaction with other teams working on the same project, and working with a large existing codebase (Begel & Simon, 2008b).

Begel and Simon (2008b) observed technical issues such as: correct use of the revision control system, and lack of robust testing due to not having access to the environment in which tests were executed. The use of Visual Studio's (Microsoft's development environment) debugger to execute breakpoints, a key debugging technique, was seen as critical during the debugging tasks; some novices demonstrated advanced technique and could navigate through the large codebase without referring to documentation, they reported having been "clued in to certain debugging techniques" (Begel & Simon, 2008b, p. 228). The other novices struggled with documentation to find the part of the code to debug; frequently taking a long time to find information, and when information was discovered they would often find it was out dated (Begel & Simon, 2008b). The technical difficulties were often seen alongside collaboration and orientation problems (Begel & Simon, 2008b).

Novice software developers have large challenges around cognition; an enormous amount of information was presented to new employees, collecting and organising this information so it was useful later on was a difficult task; some novices emailed themselves notes, others extracted important task related information from emails into other documents; sometimes impromptu teaching sessions would occur when a co-worker was approached for information, novices found taking notes difficult in these situations; there is a reluctance to press for more information if the novices are not following an explanation or lack full understanding when given an answer to a question, this was interpreted as them being anxious to not waste a senior worker's time (Begel & Simon, 2008b).

Orientation problems with team membership, codebase, and organisational resources were exacerbated for the novice software developers by often badly organised documentation, leading to much frustration and lack of progress; novices often did not know everyone on their team and were unsure who to talk to over particular issues; even though a mentoring program existed one novice had to request one after four weeks and ended up with a mentor who was too busy to provide quality information (Begel & Simon, 2008b).

Begel and Simon (2008a) discuss that while educational institutes endeavour to prepare graduates for industry by: teaching core computing theories that will help their learners keep apace with industry developments, and updating course content to reflect the current technical skills required by employers; there has not been enough focus on the "soft" skills required to tackle the human elements of software engineering (p. 3). They describe the human elements of software engineering as being:

> … the ability to create and debug specifications, to document code and its rationale and history, to follow a software methodology, to manage a large project, and to work with others on a software team (Begel & Simon, 2008a, p. 3).

In a typical software engineering course, a team of three to five students are given a set of stable requirements, from a 'real' customer, then design and build a software product to meet those requirements by the end of the course; the point being to simulate a "greenfield", new software development, project so students are exposed to a variety of development phases and learn to interact in a team (Begel & Simon, 2008a, p. 12). Begel and Simon (2008b) point out that this type of project does not reflect the situation that most novice developers will find themselves in, joining a large pre-existing team, as the most junior member, and spending most of their time, initially, working on debugging a large pre-existing codebase.

Van Slyke, Kittner, and Cheney (1998) surveyed employers from a variety of industries to determine the skills they thought most necessary for their IT graduate workers to possess. Employers in this sample placed less importance on specific technical skills, overwhelmingly ranking more highly the general skills of: written communication, analytical thinking, general thinking, team work, listening ability, and self-motivation (Van Slyke et al., 1998). The IT related skills that were considered most important for entry-level workers were the more general ones, such as: systems analysis and design, database concepts, general programming techniques, and systems testing (Van Slyke et al., 1998).

Young, Senadheera, and Clear (1999) asked employers to rank how important a selection of technical skills were, and to comment on areas where graduate skill sets were lacking. The top ranking technical skills were: systems analysis and design, specific programming languages, user training and support, operating systems, databases, networking, systems administration, use and evaluation of software packages, information gathering, and project management (Young et al., 1999). When asked to identify areas where urgent up-skilling was required the respondents overwhelmingly highlighted interpersonal and management skills, these included: written communication, ethical consultation, time management, team membership, presentation skills, information finding, customer service, and general people skills (Young et al., 1999).

More emphasis is being placed on software maintenance skills as the software industry shifts from having lots of new product development work to having more reuse of existing systems, adapted to suit new requirements (Kajko-Mattsson, Forssander, Andersson, & Olsson, 2002). In summarising the literature, Kajko-Mattsson et al. (2002) describe software maintenance as requiring more than half of the resources allocated to the life-cycle of a software product. Gunderman (cited in Kajko-Mattsson et al., 2002, p. 57) states that "maintenance has been viewed as a second class activity, with an admixture of on-the-job training for beginners and low-status assignments for the outcasts and the fallen". System maintenance tasks are often assigned to the inexperienced, who lack formal training in this area; consequently when they make what they deem to be small changes to code it can often lead to severe damage of the whole system (Kajko-Mattsson et al., 2002). McQuire & Randall (cited in Kajko-Mattsson et al., 2002, p. 58) assert "a highly skilled maintainer is the most important organisational asset pivotal for achieving quality software, strategic for improving maintenance and development processes, essential for remaining competitive and critical for business survival". They argue that a maintainer should be a skilled diagnostician and that it is up to educational institutes to prepare students for these roles (Kajko-Mattsson et al., 2002).

The Australian Learning and Teaching Council surveyed both employers and graduates employed in ICT positions to determine if Australian ICT curricula were developing work-ready graduates (Koppi & Naghdy, 2009). Both groups identified the following areas as being deficient in ICT graduates: communication skills, general awareness of the business environment and what is required in a job role, and problem solving abilities (Koppi & Naghdy, 2009). As well as these commonalities, employers highlighted the following interpersonal skills as needing improvement: initiative, self-management, independent learning, and planning (Koppi & Naghdy, 2009). Employers did not consider team-work as an area of deficiency, curricula appear to now be providing enough team-based experienced to enable graduates to quickly adapt to team-work in the business environment (Koppi & Naghdy, 2009).

4. BRIDGING THE GAP

Research suggests that the way to help prepare development students for the realities of their first job is to involve them in a project where: there is a real client or role play of a real client; the client requirements change or clients have conflicting priorities; students are made to work in teams; and the team works on an application with a large existing codebase (Begel & Simon, 2008a; Coppit, 2006; Dawson, 2000; Dawson et al., 1997; Hogan & Thomas, 2005; Joy, 2005). A team-based development project is a common element amongst many of the software engineering courses that have been mentioned, but few include all of these elements (Horning & Wortman, 1977; Leventhal & Mynatt, 1987; Shaw & Tomayko, 1991). The importance of a team-based development experience for software developers is such that all suggested computer science and software engineering curricula include group work components and they are insisted upon by accrediting bodies, such as the British Computer Society and the IEEE (Joy, 2005).

Providing experience of *programming-in-the-large*, as suggested by Shaw and Tomayko (1991, p. 40) in the large team project model, presents many challenges within the academic environment. Coppit (2006) describes the difficulties of implementing large projects in software engineering courses as being: there are additional management overheads for the instructor, students are not motivated by salary or benefits, instructors cannot fire non-performing students, students are not full-time in the course and have differing schedules which can impact on team progress, the project has a hard and fast deadline due to the academic end of semester, and students need to be graded individually not as a team. Coppit (2006) asserts it is necessary for the students to utilise state-of-the-art development tools and be involved in all of the different development activities so they gain experience in all of the facets of software development.

Surveying 30 software professionals in management roles, McMillian and Rajaprabhakaran (1999) found that the most important element to include in academic software projects was working with real users. Respondents to the survey made many comments on the ill-preparedness of graduate developers to work with real clients, where their requirements are often ambiguous and subject to change (McMillan & Rajaprabhakaran, 1999).

Dawson (2000) recommend that software engineering courses should contain a group project where the real world is simulated to help develop the personal skills of communication, planning, and adaptability required to work effectively in a team.

Begel and Simon (2008b) advocate for a more realistic team-based software engineering project where students: debug a large existing codebase, write additional features, interact with more senior members to learn about the code-base; are given incomplete directions around requirements and testing and are left as a team to work them out. Brechner (2003) proposes that to address the difference between academic based projects and commercial ones there needs to be a course with a multidisciplinary (e.g., coders, designers, technical writers) project team that has real users, vague requirements, and strict deadlines; a large scale development course would address the need for graduates to be able to write code that will be integrated into a larger piece of software, developers need to be able to read, modify, and debug the code of others who are working on the same project.

Moore and Potts (1994) implemented a three-quarter (one academic year) course practicum in Georgia Institute of Technology's Bachelor of Computer Science, which is undertaken during the student's third and fourth years. Dubbed the *Real World Lab*, the course involves industry and provides the students with real problems, real clients, and real deadlines which they must meet (Moore & Potts, 1994). The benefits of the project experience are described as being: students are better placed to deal with complex problems and large systems, with the real client exposing them to the fact that requirements are never unambiguous and providing them with problems that are too big to be tackled individually; students learn autonomy and responsibility by having to manage their projects, stick within the schedule, and choose the appropriate tools and methods for the given problem; students become more flexible as they realise requirements change and are required to undertake maintenance on installed applications to ensure reliability (Moore & Potts, 1994).

Inspired by the Real World Lab, Milwaukee School of Engineering introduced a three-quarter course, named The Software Development Laboratory, aimed at providing real world experience (Sebern, 2002). Software Development Laboratory is structured so many student teams work on large-scale on-going projects, with the premise that inexperienced students work on well-defined areas of an existing application and more experienced students define requirements and architecture of new systems or new features (Sebern, 2002). The Software Development Laboratory works with real clients and student use an incremental development model, with each quarter initially containing two development cycles (Suri, 2007). Suri (2007) describes the main academic challenges around the Software Development Laboratory as being: the lack of time spent on development tasks by the average student means there is a lack of progress on the software product; it is difficult to maintain student motivation and interest of real clients due to the slow progress of development; ratio of student teams to instructor means that there is a lack of mentoring; projects approved for the lab are non-critical due to the protracted development schedule, therefore client contact is often infrequent and busy stakeholders do not show much enthusiasm which is very de-motivating for the student teams; some projects require use of products and technologies instructors are not familiar with, meaning there is a lack of technical support for the students; students are unable to work out what their grades are based on the feedback they are getting during the course so they become frustrated. However, despite these challenges Suri (2007) asserts that students find the experience to be very valuable, with the ones involved in internships between their junior and senior year commenting that the Software Development Laboratory experience does indeed reflect the real world.

Research indicates that to best prepare software development students for real work they should be involved in a project where: there is a real client or role play of one, the client requirements change or are in conflict, students work in teams, and the team works on an application with a large existing codebase. The researcher implemented such a project as the learning environment for the BICT 300 level Software Engineering course.

5. THE LEARNING ENVIRONMENT

For the purpose of the study the learning environment was defined as: the physical classroom environment within which the participants would interact, including specific seating locations designed to improve team interaction; the virtual environment comprised of team based development tools and tools used for team communication; and the psychological environment involving real client interaction (or role play of the client) and team interaction. The learning environment was a specialist computer laboratory situated in UCOL's Palmerston North campus. The room contained 30 desktop computers running the Windows operating system. In the first few weeks of semester one, 2007, a feasibility study was undertaken by the researcher and students of the third year software engineering course to determine the type of software solution that would be most suitable for the Medical Imaging Technology students and lecturers. The outcome of the feasibility study was the recommendation that a mobile application be developed for student use in clinical placement and a supporting web application be developed for clinical supervisor use, allowing them to monitor student progress during placement.

5.1 Graduate Response - Iteration One

The software utilised in Iteration One was: Windows XP, Microsoft Visual Studio 2005, and Microsoft SQL Server 2000 running on Windows Server 2003. The development languages chosen were: C# with the .NET compact framework for the mobile application; ASP.NET for the web application; and Transact-SQL for the database transactions. The mobile application was designed to run on Windows Mobile 5.0 which, at the time, was the windows mobile operating system deployed on devices such as personal digital assistants (PDA). The development of a mobile application using the .NET compact framework and transferring of the data between the mobile device and the corporate SQL Server database using Remote Data Access (RDA) was new to both students and researcher alike. This led to a very collegial learning experience, with a lot of trial and error, as the group went through the steps of setting up a test network to prove the technology would work within the UCOL environment.

The graduates from the first iteration of the project, who were employed in development roles, were surveyed in 2008 to determine how useful the skills they developed in the learning environment had been to them in the workplace. Graduates were asked to rate how much they learnt about that skill during the project and how useful the skill has been to them so far in their development career

The development students were split into two distinct teams during the first iteration of the project: those that worked on the mobile application; and those that worked on building the database and constructing the ASP.NET web application. The total class size was seven, with two developers working with the researcher on developing the mobile (PDA) application and five developers working on the database and web application. From this sample a response rate of 71% was received, two mobile developers and three web developers.

The first section of the graduate survey contained questions related to the team-based development skills and working with a real-life client. In iteration one, the students were involved in a number of activities that interacted with project stakeholders other than the researcher. During the feasibility study they ran focus groups with Medical Imaging Technology students and lecturing staff to determine the core application requirements. Once technology choices had been made they presented the outcome to both the lecturer users and IT staff, in order to obtain approval on user requirements and application infrastructure. Once a working application was ready, user training was conducted and feedback on use of the device was recorded so modifications could be made. A training session was also delivered to clinical staff at the hospital prior to the first pilot test of the mobile application. Three of the five graduates reported their experience interacting with the client to be essential or very useful to them.

The second section contained questions relating to technical skills developed. The response as to how useful the technical skills had been to the graduates was varied, with the skills that were transferrable being of more use than proprietary software skills. Utilising the SQL Server database management system (using Transact-SQL query language) was the most useful of the technical skills for the web/database developers with all three graduates responding as very useful. The database skills were the most transferrable, meaning that skills learnt within this particular database management system could be adapted and applied to other relational databases. There was a large emphasis on data encryption and security for the mobile application. The UCOL network consultant specified everything be encrypted as the application was to access a UCOL hosted database. Both mobile developers felt they had learned a lot about mobile application security, one graduate had found these skills to be essential and the other graduate found them to be very useful. The concepts of data encryption would be transferable to other application development environments. Further comment on the technical skills developed was made by one of the mobile application developers:

> The MIT project gave me experience in the field of IT that I have chosen to work in.

The final section of questions related to the real-life project experience. All of the graduates felt their confidence as a developer had improved due to the experience. It is the researcher's experience that graduates who are technically competent often doubt their abilities until they are put to the test in a 'real job'. Additional comments provided by the graduates validated the confidence building aspects of the learning environment. Sample comments from two of the respondents were:

> I found that whilst doing the MIT project, fundamental skills were learnt that helped me understand Software Development in the real world

> The knowledge learnt from the MIT project aided me in helping to change the way my first employer developed software

All graduates used the project as evidence of experience on their curriculum vitae and four out of the five graduates referred to the project experience in their job interviews. A majority of the graduates who referred to the project in their interview used it as an example of experience in a particular technology. In relation to skills developed one graduate comment is of particular interest:

> The technical skills which were learnt did contribute towards my portfolio but it was those unrealised skills which played the crucial role, which I didn't realise until I was being interviewed in the industry world and started my job.

These *unrealised skills* were the team-related soft skills, which are often considered unimportant by technically-oriented graduates.

One graduate made extensive further comment regarding his interview. He was asked how he would cope in the position when he had no previous experience with the particular software development environment he would be working with. His response was:

> It is just a matter of learning syntax and the ambiguity of learning different software is not new to me. I guess if it wasn't for MIT [project] I wouldn't have been able to answer that question with that confidence. I was glad that there was a real example I could provide which added to my credits.

When asked if there were any skills that were required of them when they began their first ICT role that could be learnt within the classroom three graduates mentioned web related skills and another responded with a database related skill that had already been introduced into other courses in the BICT curriculum (JavaScript, JQuery, LINQ, and XML). The final skill gap identified was source control; the team-based project was the ideal place to address this skill.

5.2 Graduate Response - Iteration Two

Based on the identified skill gap, the researcher selected Microsoft Visual Source Safe for source control. At the time, this was the recommended source control solution when using the Visual Studio development environment. Changes made to the learning environment based on the feedback and due to technology updates, meant the following software was utilised in the second iteration of the project: Windows XP, Microsoft Visual Studio 2008, Windows Mobile 6.0 using Compact edition database 3.1, Microsoft Visual Source Safe 2005, and Microsoft SQL Server 2005 running on Windows Server 2003. The development languages were: C# with the .NET compact framework V2 for the mobile application; ASP.NET for the web application; and Transact-SQL for the database transactions.

The cohort of students in the second iteration (semester one, 2009) was again small, with only five enrolled. Of the five, four were invited to participate in the graduate survey. The student who was omitted from the sample had previous software development experience in industry. Of the four invited, three of the graduates responded and completed the survey. The other key element that changed for the second iteration of the project was that there was now an application with a large existing code base that had undergone a user pilot test. Students would have the experience of orientating themselves in the existing application infrastructure before they would be ready to make any changes to the application in response to issues that were raised by users during the initial pilot test. There were also not enough students to utilise peer-programming techniques, so the researcher opted for putting in place a discussion forum for team-based communication rather than focusing on agile development techniques and SCRUM which were utilised in iteration one. All students in this cohort were involved in maintenance and update of both the mobile application and web application.

Visual Source Safe provides a central code repository and simple version control that notifies if conflicting versions of the code are being ‘checked-in’. One of the graduates felt they had learned a lot about source control, one felt they had learned the basics and one responded they had become vaguely familiar. Two of the graduates reported these skills to be very useful or essential, with only one responding that they were only used occasionally

The participants in this iteration worked on separate parts of the application simultaneously. It was agreed by the group that they would restructure the mobile application so it was more maintainable in the future. The main aim of the restructuring was to separate the code that delivered the user interface from the code that contained the logic to process and store the data. This initial process of restructuring the application required a high level of communication among the group as all of their individual parts were going to be affected. The student who worked on the restructuring thoroughly documented the process within the discussion forum so other participants could alter their parts of the application to fit the new structure. Two of the graduates felt they had learned a lot about use of discussion forums for team communication, one responded as having learnt the basics. Two graduates reported this skill to be essential, one felt that it had been moderately useful.

Graduates were asked to rate how much they learnt about team dynamics, examples provided were: communication, priority negotiation, and conflict resolution amongst team members. Two responded they had learned a lot and one that they had become functional. One graduate reported these skills to be essential, the other two responded they were very useful. The final aspect of the non-technical skills was orientation within a large existing codebase. All graduates reported these skills to either be essential or very useful.

Of the technical skills, Graduates found ASP.Net and application security, and SQL Server to be the most useful. Less development was done on the backend database during this iteration as the core structure and stored procedures that manipulated the data were already in place from the first iteration and were working well, consequently the developers in this iteration felt less confident they had learnt a lot about this skill. Further comment by one of the graduates affirmed the usefulness of working on a mobile application:

“I found it useful to create an application for a mobile platform. There were functional differences and limitations imposed by the OS and compact database not encountered in the desktop forms/browser applications I had been taught up to that point”

The final section of questions related to the real-life project experience. Like the first iteration of the project all of the graduates felt their confidence as a developer had improved due to the experience. One comment made was of particular interest to the researcher. A graduate who had been employed to work in a development team for a large organisation spoke of reflecting on the project to help him maintain perspective when his confidence was flagging:

> Six months after being accepted by [my employer] when I was feeling overwhelmed by the complexity, size, scope of software and processes at [my workplace], coupled with feeling inadequate when compared to the experience, skills, knowledge base of peers, I sat myself down and upon reflection discovered that I was in essence working with the MIT project core architecture. The capturing of user data, transporting the data through different software layers to remote storage, the use of web applications to manipulate the data. The MIT application also taught code reviews, peer reviews, and team collaboration. From my experience The MIT application most closely represents the real world, be it on a smaller less complex scale. I wish I had paid more attention to some of the processes. 18 months later I still think back to the MIT application to help maintain perspective.

Two of the graduates used the project as evidence of experience on their curriculum vitae. One of the graduates referred to the project experience in their job interview as evidence of use of a particular technology. A further comment made by one graduate further validated the usefulness of the team experience:

I found that the MIT app gave a good insight into the way software is developed in a team environment, how all the parts can be worked on separately and bought together for testing or release.

Only two skill gaps were identified for this iteration. One graduate expressed a need to utilise Microsoft Team Foundation Server, a more advanced form of source control that allows for reporting on project progress as well as the central code repository and version control. The other graduate commented they would like to learn more about estimation of resources required to complete a task. Estimation is difficult for even experienced development teams; it is a skill that develops over time. The agile method SCRUM (not used in iteration two of the development project) makes estimation easier as the development cycles are short and estimation just has to be made for two-three weeks of development at a time. The researcher decided to re-introduce this method for the next iteration of the project, irrelevant of how many students were enrolled in the course.

5.3 Iteration Three - Observation

The researcher implemented Team Foundation Server (TFS) and added Team project support to Visual Studio 2008 for iteration three (semester one, 2010). This involved an update to Visual Studio 2008 to allow access to the Team Foundation Server tools through the development environment. Team Foundation Server had additional tools available to support the SCRUM development method so these were added to the basic configuration. The learning environment now had a tool which provided team web portals where work tasks could be itemised, assigned, and discussed as well as source control and conflict resolution. This allowed a more streamlined process for team communication, rather than having to use another environment to discuss work to be completed (i.e. the group discussion forum). As well as these additions and updates, the existing code was even more complex after the restructuring of the application. This would provide additional challenges around orientation within existing code.

For this iteration there were 21 students enrolled, with 19 students volunteering to be involved in the research project. Due to the added complexity of the application and the larger cohort of students (five development teams) the researcher decided to issue a ‘rescue’ card to each group. The rescue card, labelled “Call in the chief architect”, could be used once during the semester when the group felt as though they were stuck and needed assistance from the researcher. This was to aid the researcher in identifying the groups that were struggling, something that is easy to observe when working with ten or less students but not so easy when the class size is over 20.

Issues and problems that were mentioned in the student’s reflective journals, or observed by the researcher, during this iteration were categorised using the following coding frame: Team-based development environment, SCRUM development method, Orientation of large existing code base, Team conflict, Time management. The use of TFS caused the most problems for this cohort of students, with orientation within the existing application code the next most frequently mentioned issue. The complexity of TFS was mentioned throughout the semester in the reflective journals. Orientation within the existing code base was mentioned frequently in the initial weeks of the project. A comment in one student journal stating: “it’s like reading a book in another language”. All teams appeared to have a good understanding of what was required in terms of the Agile method SCRUM. All teams were observed performing daily stand-ups, the initial meeting held at the start of each development session. Peer programming was embraced by a majority of the students with the more confident developers in the team ‘buddying up’ with developers who were less sure of their abilities. The researcher observed two of the development teams getting stuck on one coding task for the last three weeks of semester. This has been recorded under time management issues as the teams that became stuck ran out of time to demonstrate complete features.

In the final week of semester the researcher held a project post-mortem. The two teams who appeared to be stuck were asked why they did not use the recuse card that was available to them. Both of the team’s responses were similar, they did not want to use up their one chance for help because then it would no longer be there. One team opted to not use any TFS features other than the source control aspect that was demonstrated by the researcher at the beginning of the development project. When questioned at the end of semester about this the group explained they felt the tool was too complicated and spending time making changes to the code was more important than using TFS. When asked if they had accessed any of the support materials provided to help them become familiar with TFS they advised they had not. Only one team made full use of the features available to them within TFS, when asked about their experiences with using the tool it turned out that one group member had put in some effort outside of class to become familiar with the tool and had been almost solely responsible for the additional use.

Based on the observations and student reflections, the researcher devised strategies to help students in future iterations of the project. The first strategy involved changes to the development project assignment task. The mark allocation and wording of the assessment was altered to move the emphasis from code fixes to use of the team-based development tools and techniques. Additionally, the researcher created a screencast demonstrating the source control aspect of TFS. There were multiple comments in the reflective journals regarding the initial demonstration of TFS, many of the students felt overloaded with information in that session and struggled to follow along with the instructions. The ability to review the material that was presented in class after the initial introduction session will alleviate this issue. There was also a noted reluctance to use the external resources provided by the researcher to assist with learning this tool, the information is complex (as is the tool) and takes some time to work through. Finally, it was decided to issue teams with two rescue cards to address the reluctance to use their “one chance for help”.

6. CONCLUSION

This study validated the concept that real-life, team-based software development projects do indeed help to bridge the gap between the academic world and what is required of junior developers in industry. The skills developed in the project, and in particular the soft skills, were found to be very useful by the graduates once they entered the workforce and they have used the experience as evidence during the job seeking process. Based on the positive outcome for the graduates of this study, the researcher is investigating the restructuring of the software development courses within the BICT degree to provide a more sustained real-life project experience.

7. REFERENCES

Ardis, M., & Ford, G. (1989). SEI Report on Graduate Software Engineering Education (1989). Retrieved from http://www.sei.cmu.edu/library/abstracts/reports/89tr021.

Bauer, F. (1973). Software and Software Engineering. SIAM Review, 15(2), 469-480. Retrieved from http://epubs.siam.org/doi/abs/10.1137/1015067.

Begel, A., & Simon, B. (2008)a. Novice software developers, all over again. Paper presented at the Proceeding of the Fourth international Workshop on Computing Education Research, Sydney, Australia.

Begel, A., & Simon, B. (2008)b. Struggles of new college graduates in their first software development job. Paper presented at the Proceedings of the 39th SIGCSE technical symposium on Computer science education, Portland, OR, USA.

Borenstein, N. S. (1992). Colleges Need to Fix the Bugs in Computer-Science Courses. The Chronicle of Higher Education, 38(45), 2-B3. Retrieved from http://search.proquest.com/docview/214642018?accountid=10382.

Brechner, E. (2003). Things they would not teach me of in college: what Microsoft developers learn later. Paper presented at the Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Anaheim, CA, USA.

Coppit, D. (2006). Implementing Large Projects in Software Engineering Courses. Computer Science Education, 16(1), 53-73.

Dawson, R. J. (2000). Twenty dirty tricks to train software engineers. Paper presented at the Proceedings of the 22nd international conference on Software engineering, Limerick, Ireland.

Dawson, R. J., Newsham, R. W., & Fernley, B. W. (1997). Bringing the `real world' of software engineering to university undergraduate courses. Software Engineering. IEE Proceedings, 144(5), 287-290.

Dijkstra, E. W. (1968). Letters to the editor: go to statement considered harmful. Commun. ACM, 11(3), 147-148. doi:10.1145/362929.362947.

Duggins, S. L., & Thomas, B. B. (2002). An historical investigation of graduate software engineering curriculum. Paper presented at the Proceedings of the 15th Conference on Software Engineering Education and Training, 2002.

Freeman, P., Wasserman, A. I., & Fairley, R. E. (1976). Essential elements of software engineering education. Paper presented at the Proceedings of the 2nd international conference on Software engineering, San Francisco, California, USA.

Freeman, P. (1987). Essential Elements of Software Engineering Education Revisited. Software Engineering, IEEE Transactions on, SE-13(11), 1143-1148. doi:10.1109/tse.1987.232862.

Ford, G. (1994). Progress Report on Undergraduate Software Engineering Education, A. Retrieved from http://www.sei.cmu.edu/library/abstracts/reports/94tr011.cfm.

Hilburn, T. B. (1997). Software engineering education: a modest proposal. Software, IEEE, 14(6), 44-48. doi:10.1109/52.636650.

Horning, J. J., & Wortman, D. B. (1977). Software Hut: A Computer Program Engineering Project in the Form of a Game. Software Engineering, IEEE Transactions on, SE-3(4), 325-330. doi:10.1109/tse.1977.231151.

Hogan, J. M., & Thomas, R. (2005). Developing the software engineering team. Paper presented at the Proceedings of the 7th Australasian conference on Computing education - Volume 42, Newcastle, New South Wales, Australia.

Joy, M. (2005). Group projects and the computer science curriculum. Innovations in Education and Teaching International, 42(1), 15-25.

Kajko-Mattsson, M., Forssander, S., Andersson, G., & Olsson, U. (2002). Developing CM3: Maintainers' Education and Training at ABB. Computer Science Education, 12(1-2), 57-89. Retrieved from http://www.tandfonline.com/doi/abs/10.1076/csed.12.1.57.8212

Koppi, T., & Naghdy, F. (2009). Managing educational change in the ICT discipline at the tertiary education level: Final Report. Retrieved from http://www.olt.gov.au/resource-managing-educational-change-ict-discipline-uow-2009

Lethbridge, T. C., LeBlanc, R. J., Sobel, A. E. K., Hilburn, T. B., & Diaz-Herrera, J. L. (2006). SE2004: Recommendations for Undergraduate Software Engineering Curricula. Software, IEEE, 23(6), 19-25. doi:10.1109/ms.2006.171

Leventhal, L. M., & Mynatt, B. T. (1987). Components of Typical Undergraduate Software Engineering Courses: Results from a Survey. Software Engineering, IEEE Transactions on, SE-13(11), 1193-1198. doi:10.1109/tse.1987.232869

McMillan, W. W., & Rajaprabhakaran, S. (1999, 22-24 Mar 1999). What leading practitioners say should be emphasized in students' software engineering projects. Paper presented at the Proceedings of the 12th Conference on Software Engineering Education and Training, 1999.

Moore, M., & Potts, C. (1994). Learning by doing: Goals and experiences of two software engineering project courses. In J. Díaz-Herrera (Ed.), Software Engineering Education. (Vol. 750, pp. 151-164): Springer Berlin Heidelberg. Retrieved from http://dx.doi.org/10.1007/BFb0017611

Sebern, M. J. (2002, 2002). The software development laboratory: incorporating industrial practice in an academic environment. Paper presented at the Proceedings of the 15th Conference on Software Engineering Education and Training, 2002.

Shaw, M., & Tomayko, J. (1991). Models for undergraduate project courses in software engineering. In J. Tomayko (Ed.), Software Engineering Education. (Vol. 536, pp. 33-71): Springer Berlin Heidelberg. Retrieved from http://dx.doi.org/10.1007/BFb0024284

Suri, D. (2007, 10-13 Oct. 2007). Providing "real-world" software engineering experience in an academic setting. Paper presented at the Frontiers In Education Conference - Global Engineering: Knowledge Without Borders, Opportunities Without Passports, 2007. FIE '07. 37th Annual.

Van Slyke, C., Kittner, N., & Cheney, P. (1998). Skill requirements for entry level IS graduates: A report from industry. Journal of Information Systems Education, 9, 7-11.

Young, A., Senadheera, L., & Clear, T. (1999). Knowledge skills and abilities demanded of graduates in the new learning environment. Paper presented at the 12th Annual Conference of the National Advisory Committee on Computing Qualifications, Dunedin, NZ.