CREATING A TEACHER AND SOFTWARE-DEVELOPER PARTNERSHIP

J. Pollard, R. Duke

ABSTRACT
As teachers are the educational experts, if we want to produce software that is both directly relevant to their needs and can be successfully integrated into the classroom, we need to facilitate a strong co-operative working partnership between the teacher and the software developer. This paper looks at a case study where we create such a partnership and study the consequences. The overall aim was to enable the cooperative teacher to play a central role in the co-designing of a mathematical educational computer program. The study focused on exploring the issues surroundings the role of the teacher and the software developer in the co-designing process, with special emphasis on the expectations of both parties. Some of these issues include:

- techniques for facilitating the sharing of ideas between the teacher and software developer;
- identifying techniques and problems when trying to express the task/aims of the software;
- identifying miss-understanding of vocabulary between the teacher and software developer;
- contrasting formal with informal lists of criteria about expected outcomes of the software;
- exploring the use of interface screen shots in ideas development;
- understanding the consequences of assumptions about equipment;
- evaluation techniques.

After preliminary discussions the cooperative teacher selected the general area of how to convert between metric units of measurement as the teaching focus of the proposed software. The responses of the co-operating teacher, other teachers, parents and students was recorded and analysed, highlighting the strengths and weaknesses of both the resulting software and the development process.

KEYWORDS
Co-operative software development, computer-aided mathematical education, requirements engineering

INTRODUCTION

To advance software design we need to look at techniques to incorporate skilled experts from diverse backgrounds into the design process. No longer can we expect software designers to have the required domain specific expert knowledge to design software in any field. Instead we should be looking at how to develop techniques to allow software engineers to actively work with experts in the area for which they are designing the software. For example, how does a software engineer co-develop educational software with a teacher? What tools do they need to make this partnership work successfully? How do their roles change/develop? What are the expectations on both sides of the partnership? From where do common miss-understandings stem? How can they be avoided? And finally, does such a partnership add any extra value to the software produced?

These questions are the driving force behind a larger project. The aim of this paper is to report on a case study which focuses on exploring the issues surroundings the role of the mathematical teacher and the software developer in the co-designing of software to help teach mathematical concepts to high school students. This paper will address the above questions in the context of this case study.

THE DESIGN PHASE

The Design Process

In this phase of the case study, the type of software package to be developed and the development strategy to be adopted had to be decided. First a volunteer teacher had to be found to take part in this experiment. The teacher who volunteered was a secondary school teacher who was head of the mathematics department in a Queensland State High School. The software to be developed was to be targeted at students in Australian Year 8 (children aged 12 to 13).

Deciding on the exact area of mathematics to be targeted by the software was the first step of the design phase. The teacher suggested areas which she believed could best benefit from the use of a computer. These were initially defined as the disciplines of statistics, graphing and elementary geometry. The criteria for deciding the mathematical area for the study was refocussed to concentrate on those areas which students traditionally find difficult. There were two main reasons for this refocusing. Firstly, the areas originally identified were very broad and would require comprehensive software to cover the multiple concepts required to teach the subject area; this would be beyond the scope of a short term case study. Secondly, the project aimed to target areas of mathematics which have not commonly been targeted by existing software packages. (Statistics, graphing and geometry are three of the most popular areas for mathematical software (AcaStat Software; Cabrilog; Microsoft; Wolfram Research).)

With reference to the curriculum for that year level, it was decided to focus on the area of unit conversion[1] as the objective for the software to be developed. This area was selected as it had been identified as an area with which students struggled both in that year level and in subsequent year levels at both a conceptual and applied skills level.

Having now defined the software's purpose, the exact way in which the program was to tackle the process of teaching both the concept of conversion and the techniques associated with converting length, area and volume had to be addressed. This process involved both the software developer and the teacher. The software developer presented the teacher with a series of screen shots taken from commercially available software products (Duality Software; Math Terra; West, 2001). The software developer also presented existing lesson plans (Terry, 2000) and exercises from currently used textbooks (Peard, ect, 1992; Williams, ect, 1973) to help suggest ideas about how to teach the specific area.

The teacher examined the various methods of representing problems using computers and identified design features which she felt were either good or bad. The software developer tried to identify why the teacher felt this way about the interfaces, through open ended questioning. Although there are tools to enable students to simply convert between units, the software developer found no existing interface that was acceptable to actually teach the concept. Hence a new interface needed to be developed.

The teacher was prompted to try and identify the ways in which these problems have been taught using traditional teaching resources and was asked to demonstrate them. A lesson plan of how to teach the concepts was then developed by creating a rudimentary story board that progressed through all the key activities required to give the students a well rounded understanding of the concept of converting units. The layout for the activities was designed around how the teacher believed the students would set out their working in a normal teaching environment. This provided the design information needed by the software developer to commence work on prototypes for the teacher's evaluation.

Techniques for sharing ideas between teacher and software developer

Once the main objective of the software had been determined, the next step for the software developer was to conduct individual research into the area of concern. In this case study this was done through the collection of both existing programs (Duality Software; Math Terra, Inc.; West, 2001) *and* relevant

[1] Unit conversion being defined as the concept of changing between metric units, e.g. 5 cm^2 converts to 500 mm^2.)

teaching aides, such as text books and existing lesson plans (Peard, ect., 1992; Terry, 2000; William, ect. 1973). The collection of these resources allowed the developer to not only consider the "look" of the program in its end form but also the way the user (i.e. the students) would interacted with the interface.

In this case the lesson plans and text book outlines enabled the teacher to point to and identify specific techniques which they wished to see implemented, allowing the "language barrier" between the teacher and the software developer to be bridged and helping eliminate misunderstandings over program content. Having already identified many ways in which the software may be designed, from a content perspective, prior to the meeting, the software designer was also able to provide useful and relevant ideas on how the subject may be better represented using a computer.

The key element to the above argument is that both the teacher and the software developer had an understanding of the content which needed to be taught and so they were both able to generate meaningful and useful dialogue on the subject. They were able to clarify ideas using examples and analyse other ways of approaching the problem. This is opposed to either the developer being explicitly told how to do it by the teacher, or the teacher being told how it must be designed by the developer. This allowed the software developer to transcend the specific knowledge area of software design and assist the teacher to think about teaching the task in different ways, without encroaching on the teacher's expertise.

As a consequence, the teacher was encouraged to consider new teaching techniques and to justify why they felt the techniques would not be as successful. These provided the software developer not only with a deeper understanding of the teacher's beliefs about teaching, but also identified techniques the teacher would not like to see implemented in the design. This information proved exceptionally useful during the design phase.

Techniques for capturing the task/aim of the software

The initial approach to defining the problem was to have the teacher attempt to write an aim or objective of what the software was meant to teach. This was a failure as the teacher was unable to articulate what exactly they wished to occurred. The teacher also became defensive at their own inability to express what they considered a common or simple idea.

To alleviate this problem instead of asking for a formal specification of the task the teacher was asked to instead provide a demonstration of how the task could be completed. The teacher then provided worked examples of how they expected the problem to be approached and tackled in a normal working environment. This not only gave the software developer the aim of the software program but also a better idea of what the user requirements would be, as mentioned above. Analysis of this working was used latter in the designing of the software, (See (Pollard and Duke, 2002b) for more details).

Vocabulary misunderstanding between teacher and software developer

The importance of clearly expressing one's perspective is essential to any discussion of ideas. Sometimes it's hard to remember what life was like before you learnt all the jargon in your field. Every field has special words used to describe specific concepts within the domain. Often it can be a steep learning curve into this area-specific domain knowledge. However, once one is emersed into the culture of the area, it becomes second nature that you use the words and expect those around you to know what they mean. This is a major problem with trying to form a strong, open partnership between a software developer and teacher. The terminology in both fields is very diverse and both parties can quickly feel threatened by the use of technical terminology from either of the domains. For example, the use of the word "bat file" or "exe" caused immediate friction, when used in a sentence like, "I'll make you a bat or exe file to run the program". However, when it was described as, "I'm designing it so that you just double click on an icon and the program starts", the tension dissolved. From a software engineer's point of view, it is important to provide information in a form with which the teacher feels comfortable. However, assuming the teacher knows nothing and avoiding details also causes conflict. The software

engineer has to learn the skill of knowing when to back off and re-explain points in simpler terms without patronizing the teacher.

An even more difficult problem is the misunderstanding of terms. This is harder to detect since both sides can believe they know what the word means within their context. An example of this is the simple word "program". To a teacher a program is a collection of lessons; a curriculum for a subject; an over plan for a course. However, to a software engineer a program is a collection of code to create a piece of software. It took sometime to discover that we were talking about different things. Sadly the only solution is experience, however we can prepare software engineers and teachers by warning them of the issue. In the case of the word 'program', the issue was eliminated by removing the word from our discussions. It was replaced by 'the software' or 'lessons'.

THE DEVELOPMENT PHASE

The development process

Having created a rudimentary storyboard of the main activities students would be expected to undertake, the next task was to actually design and develop the software from the teacher's specifications. The software developer analysed the storyboards and produced a goal for each activity based on the student learning outcome. To ensure that a diverse range of options was considered a draft interface was created for each goal based on each of six software categories.

1. Resource	electronic book
2. Tool	computer as a (sophisticated) calculator
3. Drill	given a question, input an answer and receive feedback
4. Worksheet	given several questions, input answers and receive feedback
5. Simulation	given a puzzle or virtual experiment, solve on the screen
6. Discovery	presented with a scenario, manipulate the environment, check solutions or request more information

[See Pollard and Duke (2002a) for a more detailed discussion of these software categories.]

The interface was designed with reference to all of the interfaces to the informal criteria garnered from the teacher during earlier design meetings. Based on this knowledge a prototype interface was developed for the teacher to review.

At the review the teacher generally liked the overall structure of most of the activities but suggest re-wording of some, as well as the expansion of the scope of others to include extension activities. At this point only minor reworking was required before a working system was provided for full teacher evaluation. The product design was then accepted and after three program iterations the software was debugged and ready to be handed to the teacher for implementation in the classroom.

Contrasting formal with informal criteria of expected software outcomes

Had this program been developed using strictly formal criteria it is unlikely that the design process would have been as rapid as it was. By utilising informal criteria, i.e. criteria gained from undirected questioning of related subject matter, the software developer was able to use a high degree of intuition about how the teacher wished the problem to be approached. This resulted in the developer being able to spend the majority of the development time on only a few possible solutions rather than having to prepare a large number of sample programs.

While this approach was successful in this case study it does have a number of adverse consequences. Had the developer initially chosen an incorrect approach, the software may well have required significant reworking. Additionally, by using such methods other possible approaches which could have produced more educationally beneficial outcomes may be overlooked. The software satisfied what the teacher wanted but may not have satisfied the educational requirement. The teacher's beliefs on how to achieve the learning outcomes were not challenged by this approach. The software developer simply catered to the teacher's specific educational philosophy. This raises the dilemma of building software

for the client, as distinct from building software to best perform the task. In any case, who is to make that judgment, the software developer or the teacher?

Exploring the use of interface screen shots in ideas development
Screen Shots were used extensively during the prototype development. It was possible to use screen shots to indicate in this case how the software would work because of the computational simplicity of the underlying material of the tasks. This technique may not be as successful for larger, more complex software systems.

The reason that this screen shot technique was so successful in this case, was because it allowed the teacher to visualise the role and approach of the software. This facilitated lively, but directed discussion; providing useful and timely feedback on the direction that the software was to take. The screen shots provided more information than could easily be portrayed by traditional software development diagrams such as UML 'user interface diagrams' (Fowler and Scott, 2000). Furthermore, screen shots offer a more intuitive style for the teacher. However, these screen shots and storyboards were translated into more formal specifications prior to programming.

Scheduling of user meetings
One of the interesting aspects of this design process was the difficulty in scheduling face to face development time with the teacher. This was due to the fact that the teacher had very little spare time available due to her teaching load. Only four meetings were able to be arranged with the teacher over the three month development period. This impacted on the development process as it meant that every meeting had to have productive outcomes. Another meeting could not simply be scheduled if results were not achieved. This was why a number of approaches, such as the use of screen shots and informal criteria were utilised during the development of this software. It could not be expected that the teacher would be able to spend the time to learn how to read UML diagrams (Fowler and Scott, 2000) or other software engineering specific techniques (Pressman, 1997).

THE IMPLEMENTATION PHASE

The implementation Process
Having completed the development phase, the software now had to undergo final testing and implementation. Initially the software was installed on a single desktop in the staffroom of the Mathematics Department at the high school. This allowed all of the teachers to view and trial the software. Of the eight teachers in the staffroom, five provided feedback. Four of them were very excited about the software and were very keen to see it implemented in the classrooms. One of the five teachers refused to consider using the software. He stated that this was because the software did not use the same problem layout, for the worked example, that he used in his classes. (His example layout was not considered clear or logical by the other teachers and raised questions about how the dissenting teacher actually taught the concept.)

The teachers then provided final feedback on the software to iron out any last minute faults. At this point the school began to install the software onto laptops for use. A major fault was found in the software; the screen resolutions were not consistent between laptops. As a result the software was unable to be used on all of the machines. Limited technical resources and equipment constraints made it impossible to change the screen resolutions. This resulted in the rewriting of the entire software package to enable multiple screen resolutions, putting the whole project behind by two weeks.

Before implementation of the software took place in the classroom the software was displayed at a parent teacher evening. The software developer was unable to attend and this made feedback on the usefulness of the software difficult to ascertain. However, the informal feedback about the software from parents, as passed on by the teacher, was positive.

The software was then implemented in the classrooms on desktop machines, which meant that the earlier resizing of the software was unnecessary. Overall performance and portability of the software had, however, been improved by the changes made. The reason for the change to desktops was due to increasing unavailability of working laptops, making it impossible to implement the software as a class set, as originally intended.

The installation of the software did undergo some problems despite a very simple installation arrangement, highlighting the lack of technical expertise and resources in the target school. Once this problem was overcome the software was utilised in the classroom by three separate student groups. In each case the software was used with one computer between two students.

Despite the assurance of formal feedback, only informal feedback was forthcoming. However, the feedback was quite positive and very detailed. The students appeared to enjoy the software and became very involved in the activities. The teachers felt that the software made a difference; higher than normal results were achieved on the end of unit testing, when compared to previous years.

As a non-intuitive subject area, previously students had achieved either very good marks or no marks at all in the unit tests. However students this year demonstrated a moderate understanding of some of the units of measure. The teacher believed that this was the result of them using the software, because the software enabled students to grasp and demonstrate the concept of how to convert units, even if they still made minor technical errors.

A suggestion for better implementation of the software was that the software should be made available to students during lunch time as an additional resource for revision, after the initial class lesson. This was because the teachers felt that one period was not enough time for some of the slower students to finish all of the activities provided by the software.

This completed the Case Study. Figure 1 show some screen shots from the software produced in the case study; if you would like to see more refer to (Pollard, 2002). (The software is available for download.)

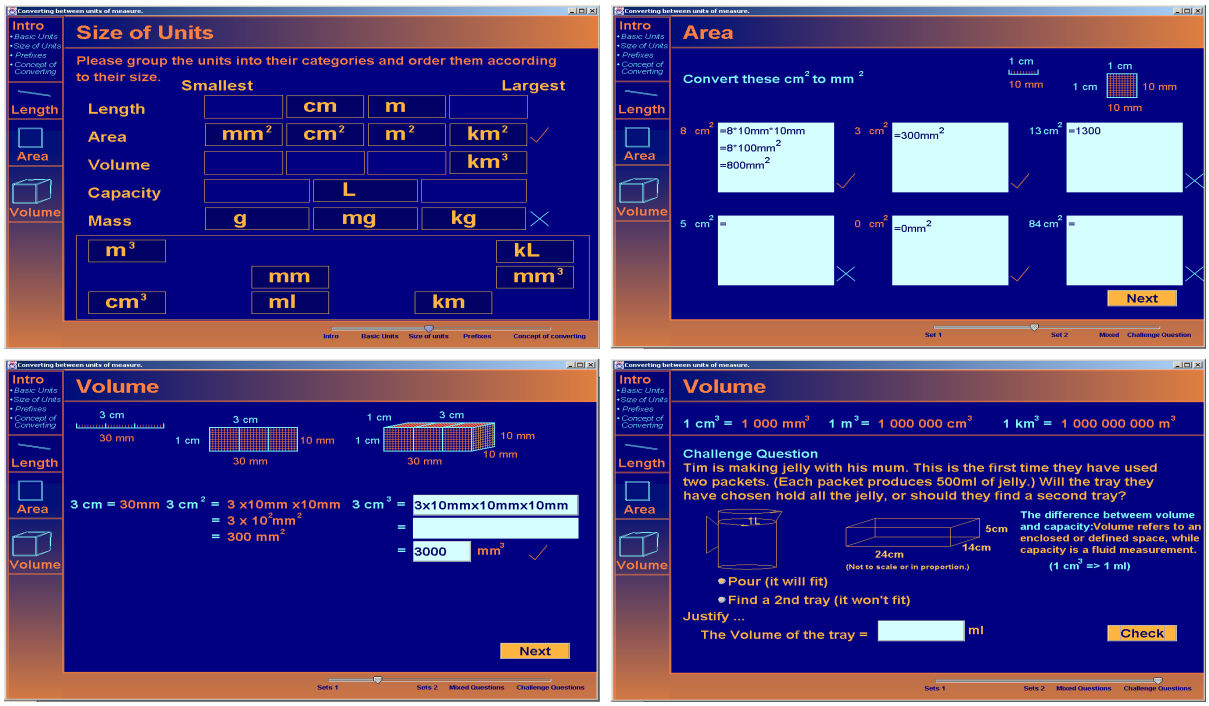

Figure1. Screen shots of 'The concept of converting'

Understanding the consequences of assumptions about equipment
Unlike a business which has a developed IT department with good technical control and understanding of their software systems, the school possessed a relatively undeveloped infrastructure. This meant that the technical specifications provided by the school were unreliable and should have been verified by the software developer prior to the commencement of the development phase. One machine specification was supplied, but given the wide and varied nature of the laptop fleet, clearly this was not indicative of all of the machines. While this may not normally fall within the realm of the developer's responsibility, the successful installation of the software depends very much upon accurate knowledge of all of the school's computer resources. This error had negative effects on the experiment by delaying the introduction of software into the teaching program.

Evaluation Techniques
One of the major factors highlighted by this experimental case study was the lack of formal feedback. All feedback received relied upon the teacher; due to legal impairments, the developer was unable to enter the classrooms. This is an area which needs to be overcome for better experimental results to be obtained. The informal feedback that was received, whilst anecdotal, did provide useful data that will form a base line for further experiments in this area. It has also highlighted a number of areas where tighter controls are necessary to improve the performance of future experiments.

Future experiments need to include the enforcement of formal questionnaires for teachers during the various phases of development and for students once installation of the software is complete.

CONCLUSIONS

The changing roles within the partnership
As the driving force behind the creation of this software, the software developer was the one who had to make the software product work and have it accepted by the teacher. This meant the software developer had to remain flexible throughout the partnership. It was essential for the software developer to understand what was to be taught and how it was to be taught. This meant that the software developer had to be able to discuss and adopt educational philosophies that were to be utilised in the software product. While this is normally outside of the domain of a software developer it was essential to the success of this partnership.

Expectations within the relationship
With regards to the software, the teacher's initial expectation of the potential of the proposed software was relatively low. Previous exposure to other IT educational products not specifically targeting elements of the curriculum had created scepticism about the worth of software products and the potential of computers in mathematical education. As this "tailor- made" project progressed, the teacher's expectations rose, as did the acceptance of software products. The teacher also demonstrated great concern over the specifics of the program and paid a lot of attention to detailed wordings and lay-outs within the software.

The expectation of the software developer to have the software accepted was partly met. Whilst it would normally be expected that the teacher would direct a large amount of effort towards the development this was not the case. The tight time tables operated by the teacher restricted any attempt by them to spend additional time developing the software. Once the teacher accepted the software in its final form she was enthusiastic about its applications in the classroom. However, the teacher did not follow through with comprehensive feedback on its use in the classroom, in contradiction of the developer's initial expectation that feedback would be forthcoming.

Misunderstandings within the relationship
Most misunderstandings between the partners in the development process occurred because of differing cultural paradigms. The specific language sets of the two paradigms triggered these misinterpretations. The use of storyboards, screenshots, diagrams, etc. limited misunderstandings, as they were free of

paradigm-specific terminology and were readily useable by both parties. This also reduced tensions between partners as neither were forced to adopt the other's specific vocabulary (and perhaps as a consequence not made to feel a less significant part of the team).

Appropriate Tools for Educational Software Development
From the data gathered during this case study the following preliminary deductions can be made:

- The collection of teaching materials/text books/lesson plans on the area to be taught needs to be done prior to the commencement of the software design phase.
- The creation of storyboards to define the task facilitates strong interactions with a teacher. This is further enhanced by the use of screen shots later in the development phase, i.e. when prototyping layouts.
- Informal criteria can be used to bridge the communications barrier between software developer and teacher when formal criteria fail.
- Extensive trialing of software to a number of teachers provides better feedback and also is likely to encourage acceptance of new software (and possibly generate ideas for further applications).
- Informal criteria are unsuccessful in evaluation as they provide only limited feedback and do not allow for detailed analysis
- Formal definition of the education objectives of the software failed in this case due to the inability to clearly articulate the educational aims in written form
- Specifications for hardware requirements failed in this case due to a lack of technical support on the schools behalf.

The failure of formal specifications is a key indicator of the difficulties involved when developing educational software for the classroom. This lack of knowledge has been observed during other interactions with teachers and is primarily due to a lack of formal training with computers. The result of this is that any attempt to gain formal specifications from teachers is likely to result in minimal success. It is therefore advised that software developers place additional emphasis on the other techniques that have been discussed above to create a better partnership with the teacher.

Future work will focus on assessing the educational worth of the software and its wider acceptance within the educational community, so as to evaluate the impact this partnership can have on the educational value of the software produced.

REFERENCES

AcaStat Software, StatCalc 4.1. Software available at: www.downlinx.com/proghtml/195/19515.htm

Cabrilog, Cabri Geometry II. Software available at: www.cabri.com

Duality Software, A-Converter. Software available at: www.dualitysoft.com/aconverter/index.html

Fowler, M. & Scott, K. (2000) UML distilled : a brief guide to the standard object modeling language. (2nd ed) Addison Wesley.

MathTerra, Inc., Advanced Converter. Software available at: www.downlinx.com/proghtml/281/28168.htm

Microsoft, Microsoft Excel. Software available at: www.microsoft.com/office/Excel

Peard, Mowchanuk, Shield, & Partridge. (1992) Interactive MATHS. Jacaranda Press, Australia.

Pollard, J, (2002) The Concept of Converting. Software available at: www.itee.uq.edu.au/~janellep

Pollard, J. & Duke, R. (2002) From Maths Problem to Program: What's the best path? Troubling Practice. Post Pressed, Australia. pp195-206.

Pollard, J. & Duke, R. (2002) A Software Design Process to Facilitate the Teaching of Mathematics. International Conference on Computers in Education, (ICCE) Proceedings. Los Alamitos, CA: IEEE Computer Society Press. pp 906-907.

Pressman, R. (1997). Software Engineering: A Practitioner's Approach. (4th ed) McGraw-Hill Companies, Inc. USA.

Terry F. (2000) Metric Conversions. A to Z Teacher Stuff: Lesson plans. Available: atozteacherstuff.com/lessons/Metric.shtml

West, R., (2001) Convert-O-Gadget. Software available at: www.downlinx.com/proghtml/316/31654.htm

William, Glenn, Donovan & Johnson. (1973) Invitation to mathematics. Dover, New York.

Wolfram Research, Mathematica 4.2. Software available at: www.wolfram.com

R. Duke and J. Pollard
University of Queensland
School of Information Technology and Electrical Engineering
Brisbane
Australia
Email: rduke@itee.uq.edu.au and janellep@itee.uq.edu.au