The Novice Programmer's "Device to Think With"

Dermot Shinners-Kennedy
Dept. of Computer Science and Information Systems
University of Limerick
Limerick, Ireland
dermot.shinners-kennedy@ul.ie

David J. Barnes
School of Computing
University of Kent, Canterbury
Kent CT2 7NF, UK
d.j.barnes@kent.ac.uk

ABSTRACT
We present some ideas for course material for the introductory teaching of programming that are based on the principle of allowing the students to be the domain experts. The idea is that the students' familiarity with the domain of discourse will make course material more motivating, and that it will be more likely that they will be able to model the concepts and artifacts being discussed. This approach thereby seeks to scaffold the students' understanding of programming-related concepts. For reasons discussed in the paper, we have chosen mobile phone technology for this discussion, but there is no reason why the same principles should not be applied to other culturally-accessible domains.

Categories and Subject Descriptors
K.3.2 [Computer and Information Science Education]: Computer science education.

General Terms
Algorithms, Human Factors, Languages.

Keywords
CS1, curriculum ideas, mobile phones, student-centered learning.

1. INTRODUCTION
Our discipline suffers an interesting paradox. It is widely accepted that the development of a knowledge economy is hugely dependent on information technology and yet student enrolments and retention levels on courses are almost universally low. The discipline also has a serious gender imbalance. One of the reasons for these problems is that existing pedagogies and materials can be inaccessible or alienating and haven't always kept pace with cultural changes. Our pedagogy has to develop and embrace new educational thinking and practice if it is to meet the demands made of it. This is one of the reasons why we have witnessed, in recent years, manifold initiatives to make introductory programming attractive and accessible: the development of a plethora of introductory programming environments such as Alice, BlueJ, DrScheme, Greenfoot, Scratch, etc.; initiatives with robotics; advocacy of different pedagogical approaches (objects-first, functions-first, procedures-first, etc.) and curriculum ideas. Yet, for all this effort, there remain no definitive research results or consensus among the education community to identify a clear winner. At present, it seems that the most likely success factor in the initial teaching of programming remains the commitment and enthusiasm of the teacher – which, at least, is reassuring for those of us who are passionate about teaching!

SIGCSE'11, March 9–12, 2011, Dallas, Texas, USA.

2. MOTIVATING STUDENTS
Teachers are always looking for approaches and examples that will be interesting to their students. At the whole-course level, this is one of the main reasons why we see robotics, for instance, being used in various ways. The same principle is also often applied at the level of individual topics within a course. For instance, recently there was a discussion on the SIGCSE mailing list [10] about the motivational appeal of traditional recursive examples. There was some consensus that exemplars such as Towers of Hanoi and the computation of factorial or Fibonacci values lack appeal. Wirth's example of parking cars [11] was one suggestion offered to provide "some real-world characteristics" in a problem with a good recursive solution. However, the central issue is whether students will be able to understand the problem being addressed, and whether they will also see a rationale for addressing it.

Aristotle once observed that, "It is the customary that is intelligible" [1]. There is a remarkable simplicity and logic to the notion that if someone has a deep interest in puzzles and games, for example, then efforts to stimulate intellectual curiosity and inquisitiveness using puzzles and games are likely to be highly successful. For someone with no interest in puzzles and games the outcomes are likely to be less successful. This is a classic dilemma for teachers – in the context of a diverse set of interests what 'things' can be used to engage the intellectual curiosity of the students?

Results from the international Relevance of Science Education (ROSE) project [9], show that the interest profiles of boys and girls follow typical gender-related differences. For example, girls tend to be more interested in human biology, health issues, reproduction and cosmetics; boys are more interested in explosive chemicals, spectacular phenomena like earthquakes and hurricanes, weapons, engines and other everyday mechanical equipment. However, there are exceptions. Both girls and boys are interested in how mobile phones, CDs and DVDs work. Ironically, none of these key twenty-first century devices appear to receive significant widespread attention in science – including computing science – curricula.

All of the concepts introduced in a typical programming course are within the conceptual competence of a young child. Children

can make choices between alternatives, repeat a task until it is complete, search a list for something, and so on. Yet, despite the fundamental simplicity of basic computer operations, traditional approaches are loaded with attempts to develop new skills in students via the medium of unfamiliar and inaccessible scenarios. The ideas we explore here seek to build on the apparently gender-neutral area of familiar technology that the ROSE project highlights, both to scaffold and motivate the learning of introductory programming. Our view is that we should try to develop curriculum ideas around those things that are both familiar and interesting to our students. The problem for us as teachers is that Aristotle's words apply equally well to us as to our students: we are most comfortable with the things that we grew up with, learned about as undergraduates, or are excited about now; but that does not mean that they will be equally accessible or appealing to our students.

3. A DEVICE TO THINK WITH

When, nearly thirty years ago, Seymour Papert published his view of, "how computers may affect the way people think and learn" [7] he described, "[a] computer rich future, a future where a computer will be a significant part of every child's life." We are clearly now in that future. Mobile phones have more computing power than the first Apollo spacecraft that landed on the moon [3], and *virtually every child and student has one*. That computing power represents what Papert identified as an object-to-think-with, "[an object] in which there is an intersection of cultural presence, embedded knowledge, and the possibility for personal identification."

Our approach is to take the students' familiarity with (and acceptance of) the mobile phone as an ordinary everyday object, and use the seemingly limitless potential it offers to explore the world of programming. However, it is important to appreciate that this is not about teaching students to program real mobile phones; rather it is about using the familiar features of a mobile phone to enable students to understand programming concepts, and hence develop traditional programming skills within a bounded, tailored environment. The ideas discussed here are built around a software simulation of a basic mobile phone, to which the students add enhancements that serve to illustrate programming concepts.

Central to this approach is the inversion of the relationship between teaching and learning. By situating the student activities in a scenario (the mobile phone) that they are not only comfortable with but in which they are experts, the teaching and learning dynamic is fundamentally altered. Instead of the tutor introducing material and providing examples, the students explore and investigate concepts and ideas that they are already familiar with. The tutor simply provides the conditions in which the students can learn. In our view, mobile phones epitomize the type of object-to-think-with that Papert envisioned. This approach articulates a pedagogy that exploits students' familiarity with mobile phone technology to scaffold their learning of programming.

Ausubel succinctly summarized the basic idea when he wrote, "If I had to reduce all educational psychology to just one principle, I would say this: The most important single factor influencing learning is what the learner already knows. Ascertain this and teach him accordingly." [2]

There is an interesting parallel to our approach to be found in the nineteenth century. In a series of six lectures he delivered to young people at the Royal Institution in London in 1861, the renowned experimental scientist Michael Faraday chose a candle, a simple, everyday object, to explain the fundamental principles of chemistry. Faraday chose a candle because, he explained, "There is not a law under which this universe is governed which does not come into play and is not touched upon in these phenomena." [5] In a similar vein we would suggest the utility of using a mobile phone for the same purposes within the field of computing. To paraphrase Faraday, we would dare to suggest that, "There is not a law under which computing is governed which does not come into play and is not touched upon in mobile phones!"

4. CURRICULUM IDEAS

In this section, we provide a small sample of the many ideas that can be drawn from the scenario to provide material for various stages of both CS1 and later courses. While our focus is primarily object-based, the domain of the mobile phone is clearly not tied purely to an object-oriented approach. It also offers scenarios for many of the procedural and algorithmic elements we wish to teach.

4.1 Fundamentals of classes and objects

Successful introduction of object-orientation requires the establishment in students' minds of a clear understanding of the nature of classes and objects: class as a type and objects as independent instantiations of that type; objects maintaining state variables that require initialization, may be inspected and might be modified. Classes that relate to concrete artifacts from a student's experience stand the best chance of communicating these facets. An example we might use is that of a text message. We can talk about text messages in general terms: they contain characters that make up the message; they are sent to someone; they have a date and time of sending; they have a sender. All of these are familiar ideas and relate to the notion of class or type. But we can also talk about specific text messages: the one I sent to Helen this morning about meeting up after class; or the one Tim sent me yesterday asking to borrow my notes. These are the individual instantiations of the general idea of a text message. Every text message has the same components (structure) at the class level, but every individual text has its own settings for those components at the object level. Little, if anything, of the scenario needs explaining to the students, because they know this already, and from that understanding we can easily link to the new (programming) terminology we wish to use. From there we can introduce some simple code to implement those ideas.

The class in Figure 11 contains just the essential elements to represent a basic text message: fields to store the configurable items, a constructor to initialize the fields, and accessor methods to inspect the values of the fields. Since text messages are essentially immutable, we can omit mutator methods at this stage. Within an environment such as BlueJ [6] a complete project containing just this class could be defined, and instances created and investigated via its object bench – giving physical reinforcement to the ideas of instance multiplicity, identity of structure, yet individual expression.

In order to illustrate that we are not suggesting that the mobile-phone scenario offers just one route through a course, but a rich source of ideas that can be adjusted to fit the needs of both students and course material, we offer an alternative introduction to these class fundamentals. This route might be used with students who have existing programming experience, for instance, or who require a more sophisticated approach.

```
public class Text
{
    // Components of a text message.
    private String from, to, message;

    // Constructor/initializer
    public Text(String whoFrom, String whoTo,
                String mess)
    {
        from = whoFrom;
        to = whoTo;
        message = mess;
    }

    // Field/property accessor
    public String getFrom()
    {
        return from;
    }
    ...
}
```

Figure 1: Elements of a text-message class

The level at which we pitch the notion of class is completely flexible, and is simply a reflection of the degree of abstraction we wish to work with at any particular time. For instance, we might wish to focus on the phone's physical aspects and look at the whole phone as an object; or we might wish to consider its individual components as objects – the keypad, the screen, the battery, and so on. The notion of abstraction is obviously a particularly important one in the world of object-orientation, and one that will be touched on repeatedly at different times in a course. For some students, being able to relate the concepts of class and object to physical things may be easier than virtual things like text messages. With this in mind, we might define a class to model the battery of a mobile phone. A battery has a very simple state description – its level of charge – that will need to be monitored via an accessor method. We will want to modify the state via charging, and the charge level will be reduced through use of the phone. Exactly how realistically we want to try to model charge depletion and restoration is up to us – sending a text or receiving one might incur a charge cost, for instance. In one simple implementation we have used a small thread associated with a battery object that periodically reduces the level. This means that we can create a fairly realistic scenario in which a number of battery objects can be created and individually queried for their levels, which obviously vary independently. A battery would probably not have a visualization at this stage, because that is not an inherent part of its functionality but an artifact of presentation.

While the scenario is clearly more sophisticated than the one using Text objects, the lessons are still the same: the Battery class defines a regular structure that all battery objects share; the class has a field for storing a battery's level, which can be queried; and each instance maintains its state independently of every other. What both examples have in common is that students get to work with stand-alone elements that they can explore, develop and test independently, and that will later be integrated with other components to build more complex systems. That provides a further important practical illustration of the value of writing cohesive classes that don't try to do too much – sophistication is achieved through composition.

4.2 Collections

Collections are pivotal structures in programming systems. As a source of examples the mobile phone is almost unrivalled. A mobile phone is essentially a collection of collections! For example, a typical mobile phone has a contacts list that is usually alphabetically ordered (ascending); a speed dial list that is ordered on the basis of personal preference (ranking); an incoming message list ordered by time of arrival (last-in first-out); recently-used number lists that reorder dynamically; text prediction and other word lists ordered on frequency of use (self-organizing); and so on. Where audio or image files are supported, the phone allows the user to create their own collections and populate them as they please. For example, most users arrange their audio files into 'playlists' which can be played sequentially or in random order. Similarly, image files can be arranged into 'slide shows.'

Mark Twain once observed that, "It ain't what you don't know that gets you into trouble. It's what you know for sure that just ain't so!" When asked, most students will claim not to have encountered a self-organizing list, a list that uses a last-in first-out mechanism, or a list that could be empty. However, a little prompting helps them to realize that the text prediction feature in some phones initially displays word suggestions based on general usage but, over time, prefers the words that have appeared frequently in the text they have typed. How the phone handles incoming messages by placing the most recent as the first reveals the stack-based nature of the incoming message list. The possibility of an empty 'to-do' or 'missed call' list seems obvious once it has been brought to your attention but invariably evades cognition in a free recall scenario.

This type of inert knowledge is notoriously difficult to utilize and led Perkins to label it as 'troublesome knowledge' because it, "sits in the mind's attic, unpacked only when specifically called for by a quiz or a direct prompt but otherwise gathering dust." [8] Unknown to them, mobile phone users have been exposed to virtually every collection operation that would be included in a typical data structures course. They have experience of collection creation, maintenance and removal. They are aware of the benefits of having a variety of access strategies for collection elements (e.g., browsing, search, random selection, iteration in specified sequences). In addition, the idea of using two different 'keys' to locate entries in a single collection is not a novelty. For example, when they receive a call the phone software searches their contacts list for the number and, if found, displays the associated name on the screen. This useful feature allows them to decide whether to answer the call (i.e., it's a friend calling) or not (i.e., it's a parent calling). Conversely, when they are making a call they can search the contacts list for a particular name and have the software dial the number of the chosen entry.

As the earlier enumeration of different possible organizations highlighted, collections are a rich source of conceptual exemplars ranging from the very simple to the very sophisticated. They also encapsulate a wide variety of simple but significant algorithms

that the typical programmer uses and reuses in the course of developing a system. Many of these algorithms are provided in libraries and other support tools but their exposition to novice programmers can help them perfect their design and implementation strategies and skills. Figure 22 is an example of using a list structure that many phone owners will have used unconsciously in a variety of contexts, such as randomly playing tracks in a playlist or randomly displaying images captured with their phone. From examples such as this, parallels can easily be drawn beyond the basic scenario.

```
void playShuffled(ArrayList<Track> playList)
{
    int toPlay = playList.size();
    while(toPlay > 0) {
        int chosen = random.nextInt(toPlay);
        Track track = playList.get(chosen);
        play(track);
        toPlay--;
        // Swap the chosen track with the
        // last one left to play.
        playList.set(chosen,
                     playList.get(toPlay));
        playList.set(toPlay, track);
    }
}
```

Figure 2: Method to play tracks in a playlist randomly

As Figure 3 shows, with minor alterations the algorithm can be used to randomly choose `n` unique numbers in the range `1..max`. (For the sake of variation we have used an array this time rather than a list.) This is another algorithm the equivalent of which many will have exploited, again unconsciously, if they are in the habit of playing the lottery.

```
void chooseLotteryNumbers(int n, int max)
{
    // Create the set of possible numbers.
    int numbers = new int[max];
    for(int i = 0; i < max; i++) {
        numbers[i] = i+1;
    }

    // Select and print n from the set.
    int remaining = max ;
    for(int i = 0; i < n; i++) {
        int chosen = random.nextInt(remaining);
        int selectedNumber = numbers[chosen];
        System.out.println(selectedNumber);
        remaining--;
        // Swap the chosen number with the
        // last one left available.
        numbers[chosen] = numbers[remaining];
        numbers[remaining] = selectedNumber;
    }
}
```

Figure 3: Method to select n unique values from 1 .. max

4.3 Exception Handling

Most mobile phone users have had the experience of their battery going 'dead' or 'flat' during a call. Similarly, phone users operating on 'pay as you go' or 'top-up' schemes have tried to make a call or send a message only to discover that their credit balance is insufficient to cover the cost or runs out during the call. These experiences will have made them aware of the fact that the severity of the problem will influence both the amount of thought given to its likely occurrence and the effort required to resolve it if it does occur. For example, they know that if they are unable to receive messages because their message box is full the problem can be resolved quite easily by deleting some messages. The problem is a trivial one, as is the solution. A flat battery is more significant. Resolving that problem requires possession of a phone charger and access to a power source. In the absence of either or both of those resources the problem appears unsolvable, unless you are a creative mobile phone user who understands that convincing someone to let you put your SIM-card in their phone keeps you 'online'. In the same vein, running out of credit is a very difficult problem to solve if you have no money. While friends may allow you to run down their battery, they may not be quite so willing to let you run down their bank balance! If you have no credit and no access to funds you are 'offline' – period.

Simple problems can usually be handled easily. For example, putting the same name and number in your contacts list may be a nuisance but nothing more significant. Harder problems may require more work and cooperation with other entities, or objects. Serious problems need to be checked out and acted upon otherwise the system may be compromised. Some problems cannot be resolved and oblige us to stop using the phone.

The fact that the foregoing appears painfully obvious is precisely why it is pedagogically attractive to tutors. Phone scenarios provide a considerable range of contexts for exploring the nuances of exception handling. The following example is but one of very many candidate examples.

Mobile phone companies measure numbers of text messages handled in billions per month and it is not uncommon for young mobile phone users to send *several hundred* text messages a day! As a consequence they are very familiar with the types of difficulties that can arise and those difficulties (literally) need no introduction. For those of us lacking that expertise the difficulties can include:

- unknown number (i.e., recipient);
- a weak or non-existent signal;
- insufficient credit;
- recipient cannot be reached at the moment because of their signal level;
- network difficulties (i.e., the provider's infrastructure is not fully operational).

The outline of a possible message-sending method is shown in Figure 4. It is passed the recipient's number and the message text. It uses the Text class of Figure 11 to encapsulate the details required by the provider transmitting the message. The provider attempts to transmit the message. Success produces a confirmation message on the phone screen. Failure results in an exception being thrown by the provider.

The method handles only those exceptions it is competent to deal with. Cases involving an unknown number or poor signal are propagated up the method-call stack and left to be handled at the appropriate level. For example, the code used to choose a number, or allow one to be entered, can catch the unknown-number exception and provide the user with an option to supply another number. It may even impose a limit on the number of attempts the user can have. Handling exceptions involving poor or no signal may include an option to move the phone before trying to resend.

```
void sendMessage(String whoTo, String msg)
  throws ...
{
    try {
        Text txt = new Text(number, whoTo, msg);
        provider.sendText(txt);
        screen.setStatusBar("Message sent.");
    }
    catch(InsufficientCreditException e) {
        // Put txt in DRAFTS ...
    }
    catch(UnreachableException e) {
        // Handle exception ...
    }
    catch(NetworkProblemException e) {
        // Handle exception ...
    }
}
```

Figure 4: Sending a text-message with exception handling

Readers will, of course, note that many additional features could be added to the scenario and that, in many respects, it is a simplistic exemplar of what is actually required to handle the possibilities that arise. Fortunately the same observations will be made by most experienced mobile users. While the notation may be unfamiliar to them, most phone users can, without prompting, quickly spot deficiencies in the scenario and, more importantly, offer suggestions as to how it might be improved. Students trade their inadequate knowledge of the mechanism used to specify how to achieve something for their expert knowledge of the things that need to be achieved. Their status as collaborators allows them to focus on acquiring what they don't know (i.e., exception handling mechanics and the associated notation) and contributing what they do.

The typical mobile phone user is fully aware that the exceptions enumerated in this example are not unique to the task of sending text messages. In those circumstances the idea of developing a classes of error conditions that would be beneficial for all operations associated with the phone is quite sensible.

One of the key attractions of discussing exception handling in the context of a mobile phone is that you cannot afford to ignore them, or avoid them occurring. Maintaining the integrity of the system by correctly handling exceptions is a critical aspect of a phone's operation. This feature is rarely the case in most example scenarios used to teach students about exception handling. Having the software in your phone 'crash' simply because you mistyped a recipient's number would be completely unacceptable. Broader discussions about software features such as quality, robustness and safety are facilitated by the students' own evaluations of their mobile phone implementations. The centrality of good design is readily identifiable.

5. A CONCEPTUAL MISCELLANY

In its role as a 'device to think' with the range of concepts encapsulated within the mobile phone is truly extraordinary. In the preceding section we considered three examples at some length. In this section we offer a further series of conceptual 'snapshots' as a sample to highlight the diverse nature of the possibilities.

If at your first meeting with a class of novice computing students you ask them if they have ever undertaken a user interface evaluation their initial response will usually be, "A what?" followed quickly by a unanimous, "No." Further investigation will probably reveal that some of the class use 'text prediction' on their mobile phones and some of them don't. Asking each group to explain why they do or don't provides an initial list of pros and cons of text prediction systems. Putting it to them that clearly they had previously undertaken a user interface evaluation, albeit in a superficial and unsystematic fashion, not only establishes the credibility of their experiences but also encourages them to reflect more deeply on activities they are about to engage in.

For most of us "RUK4DD82MWB" is an unintelligible jumble of letters and numbers. (Aside: It rarely occurs to programming tutors that their early examples may have the same status!) That it might be a meaningful representation of something is disconcerting. Yet introducing some minor formatting features may make it a little more comprehensible. For example, simply introducing some white space can make it look like "R U K 4 D D8 2M WB." The knowledge that D8 is an abbreviation for DATE, 2M an abbreviation for TOMORROW and WB for WRITE BACK might allow us to decipher the message as "Are you OK for the date tomorrow? Write back."

While the original message is perfectly understandable to a text-messaging zealot, they often complain about its 'style.' Many, especially those who use text prediction, would insist that the message be typed without abbreviations. Others find the cryptic nature of the abbreviations quite acceptable and even laudable! For a programming language tutor it is quite useful to allow differing views on the stylistic properties of text messages to be voiced because it is a topic that has to be addressed in the context of programming languages as well. Of course, the same issues will surface but the benefits of consensus in the application of stylistic conventions will appear self-evident in light of the prior exposure to the text messaging discussion.

After a number of experiences of the type just outlined, students acquire a slightly different disposition when responding to what appear to be the tutor's innocuous questions and prompts. For example, when invited to state if they have ever used a WORM (i.e., Write Once Read Many) device they tend to pause a little while longer before answering. Their altered disposition is a sign of their maturing ability to reason. They have abandoned their knee-jerk response approach and substituted in its stead a more considered evaluation of the question or problem posed. This is a desirable quality in a graduate of computing!

The WORM question may evoke an association with DVDs, CDs, credit cards, security badges and similar devices or objects. Success with this type of association has a significant impact on the student's confidence and self-belief. This type of affective outcome can be as important as any achievements associated with the acquisition of technical expertise.

The everydayness of WORM-style objects can make the concept of immutable objects seem almost mundane. The fact that, for example, in some programming languages a string can be created but not altered is viewed as a feature of strings not a flaw in the language. Immutable objects have a place in the world and by extension must have a place in any model of that world that we choose build.

Modelling the world of mobile phones invariably draws us into the management of phone charges and billing. At its simplest this affords the opportunity to explore issues of database design, information retrieval and update. In a broader context it provides an opportunity to explore the elicitation of organisation knowledge and rules, the design and specification of systems to operationalize

the findings and the implementation of a system to realize the design.

As features are added to the software implementation of a mobile phone, or design decisions made about exactly how those features will be implemented, the sense emerges that we will want to have phones with variations – in the real world, not all phones are the same. Nevertheless, students can probably identify a core of functionality that is shared among all phones, while not resulting in a complete phone. The concept of abstract classes emerges from this analysis, along with concrete subclasses. There is also a technical progression observed in the phone world: manufacturer M adds special feature F to their phone, and other manufacturers quickly follow in order to keep up; the process repeats. When thinking about this in software terms, feature F might start as a subclass feature but, for the sake of utility and good structure, eventually gets refactored from the subclass to the superclass.

Finally, how one interacts with the phone services and facilities provides plenty of opportunities to consider the challenges associated with the development of a GUI for a mobile phone.

6. EXPERIENCE

This approach has been used successfully via a pilot run in 2008 and a more developed version in 2009. We are continuing to expand the ideas further in the current academic year and ultimately plan to make the materials available to the academic community. A number of students have reported that they cannot use their mobile phones now without thinking about what is going on inside. What is particularly powerful about this outcome is that it is reinforced every day because the students use their phones every day.

As indicated in the introduction, it would be naive to suggest that the ideas outlined in this paper necessarily offer a guarantee of success. However, we do believe the mobile phone scenario has the potential to support the type of 'spiral curriculum' approach advocated by Bruner, "A metamorphic spiral in which at some simple level a set of ideas or operations were introduced in a rather intuitive way and, once mastered in that spirit, were then revisited and reconstrued in a more formal or operational way, then being connected with other knowledge, the mastery at this stage then being carried one step higher to a new level of formal or operational rigour and to a broader level of abstraction and comprehensiveness. The end state of this process was eventual mastery of the connexity and structure of a large body of knowledge." [4]

7. CONCLUSION

We have presented a sample of ideas for course material for the introductory teaching of programming that are based on the principle of allowing the students to be the domain experts. Having chosen a subject domain known to have wide appeal across both cultures and genders – the mobile phone – we make use of students' existing interest and expertise to explore a phone's features via software development. This idea mimics an approach taken by Faraday to explore the world of chemistry via the medium of a candle. However, just as a candle would be of little use for this purpose today, it is inevitable that at some point in the future the utility of the mobile phone as we have illustrated it is likely to be surpassed by some other near ubiquitous device. The essential message of this paper, therefore, is to encourage teachers to maximize the opportunity for broad engagement with their students through making use of those things that are familiar to and interesting for the students.

8. ACKNOWLEDGMENTS

This work is supported in part by a SIGCSE Special Projects Grant. The views expressed are those of the authors.

9. REFERENCES

[1] Aristotle 350 BCE. *Metaphysics*.

[2] Ausubel, David 1998. In *Learning, Creating, and Using Knowledge: Concept Maps as Facilitative Tools in Schools and Corporations*, Joseph D. Novak. L. Erlbaum Associates, Mahwah, N.J.

[3] Ben-Ari, M. 2005. The concorde doesn't fly anymore. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education* (St. Louis, Missouri, USA, February 23 - 27, 2005). SIGCSE '05. ACM, New York, NY, 196-196. DOI= http://doi.acm.org/10.1145/1047344.1047354.

[4] Bruner, J.S. 1960. *The Process of Education*, Harvard University Press, Cambridge, MA.

[5] Faraday, Michael 1978. *The Chemical History of a Candle*, Cherokee Publishing Company, ISBN 978-0877972099.

[6] Kölling, Michael, et al 2010. *BlueJ – the interactive Java environment*, http://www.bluej.org/ Accessed 2010.09.01.

[7] Papert, Seymour 1993. *Mindstorms: Children, Computers and Powerful Ideas*, 2nd edition. Basic Books, ISBN 9780465046744.

[8] Perkins, David 1999. The many faces of constructivism. *Educational Leadership* 57, 3 (Nov 1999), 6-11

[9] ROSE, *The Relevance of Science Education*. http://www.ils.uio.no/english/rose/ Accessed 2010.09.01.

[10] Topham, Dave 2010. *CSI Functions First*. sigcse-members mailing list discussion initiated 11th July 2010.

[11] Wirth, M. 2008. Introducing recursion by parking cars. *SIGCSE Bull.* 40, 4 (Nov. 2008), 52-55. DOI= http://doi.acm.org/10.1145/1473195.1473219.