DECISION SCIENCE INSTITUTE

Extracting life skills from an introductory Excel programming class

Ben Martz
Shepherd University
bmartz@shepherd.edu

David Manning
Northern Kentucky University
manningd@nku.edu

ABSTRACT

The driver for this paper is the ongoing discussion around skills gaps in the education industry; gaps that suggest what the education industry is teaching does not match up with what employers need or with what students need as life skills. Multiple studies across multiple academic fields document these skill gaps at high school, college, and post college levels. The paper proposes a set of integrated introductory programming concepts based around implicit learning and cognitive science to help address this gap. The authors document how exercises found in an introductory programming class can address the claimed deficits.

KEYWORDS: Problem Solving; Career Skills Gap; Programming Skills; Life Skills

INTRODUCTION

> 'A big gap separates what graduates offer from what most employers need' ~ Doria, et al. 2002

There is much discussion about gaps between the skills the education industry provides students and what skills students need for their jobs and careers. Versions of these gaps have been suggested and data produced at multiple levels: undergraduate, graduate, workforce, etc. Barr and Tagg (1995) define a 'gap' using two areas of concern: the 'espoused theory' and the 'theory in use'.

The gap is more fully exposed when the idea of teaching more real-world concepts, the espoused theory, is compared to what is really going on in schools, the theory in use. For example, a gap exists between what skills students think they need and what skills recruiters are recruiting for. Martz and Landof (2000) found that recruiters rated problem solving skills significantly higher than did students when considering these skills as desirable now, desirable 3 years in the future, and desirable for career advancement. The results showed gaps on multiple skills across all three measures.

LITERATURE REVIEW

There are more suggestions that the gap exists at the undergraduate level. At least two early studies, *Principles for good practice in undergraduate education* (Chickering and Gamson, 1987) and *What research says about improving undergraduate education*

(AAHE, 1996), discuss this concern as they offer suggestions for improving the undergraduate education process. A contemporary study by Educational Testing Service (ETS) (Goodman, et al. 2015), suggests that conditions are not improving, specifically for the US. While concentrating on the 16-34 year old age group, the report found that 'despite having the levels of educational attainment of any previous American generation, these young adults on average demonstrate relatively weak skills [when compared to the same age group form other nations] in literacy, numeracy, and problem solving' (preface). The cause of the rankings (15th of 21, 21 of 21, and 20 of 21) are for other discussions and debates to be argued elsewhere. For the purpose of this paper, the report clearly demonstrates another 'gap' between what is needed and what is being learned.

CIOs have been placing more and more emphasis on the job candidate's knowledge of business fundamentals (Overby, 2006). According to a Forbes magazine survey (2013), top skills that are valued by employers center on '… more about how you think systems through and work within the context of a team…' (Casserly, 2012). Pfeffer and Fong's 2002 analysis of MBA business programs produced a similar concern; an analysis that generated the distress for Doria et al. (2003) to offer the opening quote.

Other studies focus on business oriented and more specifically, information technology jobs and careers. A 2014 report by kdnuggets (a leading website for technology based jobs in business analytics) showed the top five skill-orient phrases most mentioned in job announcements as: team, business, analytics, design, and development (Piateetsky, 2014). Drilling down into information technology 9 of the top 11 IT careers for 2014 listed by *US News and World Report* (Taylor, 2014), include key business areas such as Software Developer, Computer Systems Analyst, Web Developer, Information Security Analyst, Database Administrator, IT Manager, Computer Programmer, Computer Systems Administrator, and Computer Support Specialist.

There are some common threads in the skills being requested. One of the most substantial studies of general skills required for jobs was the 2013 study by the International Data Corporation (IDC), *Skills requirements for tomorrow's best jobs* (Anderson & Gantz, 2013) In this report, IDC analyzed 14.6 million job postings in an effort to identify 1.) jobs with highest growth and wage for 2020 and, 2.) top skills required for these jobs. Their results showed that "The most required skills across all occupations include oral and written communication skills, attention to detail, customer service focus, organizational skills, and problem-solving skills (p. 6)." Three of these; oral and written communication skills, attention to detail, and problem solving skills were evaluated as "cross-functional" or skills being required by more than 50% of the high-growth / high-wage positions reviewed. The concern for a lack of emphasis on these areas is echoed by the Conference Board (Casner-Lotto and Benner, 2006) wherein their survey respondents rated teamwork/collaboration and problem solving skills as "very important" at all three levels of analysis; high school, college, and college graduates. In the more narrow business education area, Topi et al (2010) define critical thinking, problem solving, and creativity as components of the 'foundational knowledge and skills' (p. 21) of an information system professional.

Regardless of level or area, these gaps emerge from the comparison of expectations. Hall and Mirvis (1995) propose that the expectations of the roles played by educating entities and companies hiring graduates have changed. They suggest the change is based on how the typical career is now viewed as self-based versus the long-time, organizational careers of the past. In their view, a career now calls for more "learning how to learn" skills than

concentrating on specific skill retraining. One example of a changed expectation posited by Fischer (2013) is the expectation about which institution, the university or the business, is responsible for preparing a new employee.

The purpose of this paper is to indicate how some of the underlying skills found in an introductory programming course are representative of the desirable foundational skills requested; those skills which are applicable in other courses of curriculum, but more importantly, are applicable as candidates for status as 'foundational' (Topi et al. 2010), 'meta-skill' (Hall and Mirvis, 1995), or 'cross-functional' (Anderson & Gantz, 2013). The argument will be supported by examples of foundational skills students are exposed to and practice in a programming course. These foundational skills become primary skills on which more complex skill sets can build. In the end, these complex sets transfer to careers. The overriding proposition is that we are teaching more foundational skills than given credit for and further, we need to show the evidence supporting that claim.

Programming and Problem Solving

At the heart of this proposition is the argument that many of the key skills inherent in programming are problem solving skills. It is programming's proximity to the raw ability of problem solving that makes programming such a strong candidate for foundational career skills. Problem solving skills have been consistently suggested as a foundational, core skill listed across many surveys and reports previously discussed (Casserly, 2012; Anderson & Gantz, 2013; Piateetsky, 2014; Goodman et al., 2015).

In many ways, the programming activity found in introductory programming classes produces implicit learning of problem solving skills. The first way is the problem solving environment of programming. In programming one is constantly charged with going from a problem state to a goal state. A student 'picks up' many skills as they progress through ever increasing problems. Piaget (1929) envisioned implicit learning as a process of assimilation and proposed it as a base learning strategy in early childhood learning. Papert (1980) defined learning as the development of problem solving models by taking skills from one situation and applying them to another situation. This adaptation of skills from situation to situation was termed appropriation. Today, a similar idea is found in the 'transfer' activity found in the Lifelong Learning VALUE Rubric supported by the Association of American Colleges and Universities (AACU) (2015). So, we propose that the basic programming environment produces an environment conducive to and supportive of appropriation of important career skills.

The second skill deals with memory structures. Most people agree with the notion that learning a new fact or skill means it is stored in and can be recalled from long term memory. Many researchers have worked to understand the structures by which memories, and thereby, learning occurs. Schank's (1988) scripts, Thorndyke and Hayes-Roth's (1979) schemata, Sanderland et al. (1983) templates and Roby's (1966) self-enacting response sequences all represent visions of a memory structure that allows the efficient storage and retrieval of facts and information in the brain. Programs themselves act much like these memory structures.

The final factor contributing to the implicit learning of problem solving skills is the actual transference of programming skills to more generic problem solving skills through protocols, metaphors, and analogies. Newell and Simon (1972) created detailed protocols in their General Problem Solver (GPS) model. Minsky (1988) defines a metaphor as 'that

which allows us to replace one kind of thought with another'. Schank (1988) is more operational in his definition and suggests a process called analogical mappings. This is where the individual is asked pointedly to compare and contrast the current problem situation with other known problem situations in an effort to gain comparative information to find solutions.

Completing programming problems require the understanding of conceptual 'memory structures' like variables, linked-lists, functions, procedures, etc. The ability to recognize problems and recall and adapt previous solutions to the new problems through analogies operates as a recognized learning process. In the end, the programming environment provides a potentially rich active learning environment for students to learn problem solving skills.

Programming and Problem Solving Topics

If we are you use analogies as the basis for our learning strategy, we must decide on the seminal topics. To help decide what topics to build on, reviewing the proposed curricula from two reference disciplines, Information Systems and Computer Science, generated a listing of topics in introductory programming courses. Table 1 lists selected higher level programming concepts extracted from two curriculum-oriented reports: 1.) the 2013 Computer Science Curriculum Report (ACM/IEEE, 2013); specifically, the Software Development Fundamentals knowledge area; and 2.) the IS 2010 Curriculum Guidelines; specifically, the learning objectives of the suggested application development course (Topi et al., 2010) This extraction seems reasonable as the two teams of authors placed fundamental programming concepts in these course descriptions as indicated in Table 1. The Association for Computing Machinery (ACM) course description defines the topics in the course as 'a prerequisite to the study of most of computer science'. (ACM/IEEE, 2013) and the IS 2010 report provides the following learning objective: 'Students will learn the basic concepts of program design … problem solving, programming logic …' (Topi et al., 2010, p. 56). Using these two curricula areas seems appropriate. While the two disciplines ultimately focus on information technology and programming at different levels, it seems reasonable that there be commonalities in the introductory programming course from both areas.

Table 1: Fundamental Concepts for Introductory Programming Course

• The role of algorithms in the problem-solving process • Program Decomposition • Basic syntax and semantics of a higher-level language • Variables and primitive data types • Expressions and assignments • Simple I/O including file I/O • Conditional and iterative control structures • Functions and parameter passing • Arrays • Records/structures (heterogeneous aggregates)	• Modular Expressions • Procedures, Functions, SubProcedures • Variables • Literals • Types • Passing parameters • Input/Output (I/O) design • Graphical user interface (GUI) • Conditional logic • Iterative design • Data types (primitive and compound)

• Strings and string processing • Debugging strategies • Documentation and program style • Testing fundamentals and test-case generation	• Arrays • Prototyping • Application integration
Software Development Fundamentals (ACM/IEEE 2013, p167-178)	Application Development Course (Topi et al., 2010, p.56)

Proposed Foundational Learning Programming Concepts

Adopting an appropriation paradigm with a foundation in Piaget's (1929), Papert's (1980), and Schank's (1988) analogous learning models, we created Table 2 to help preview and summarize the rest of this paper. In the leftmost column of Table 2 we list programming areas along with key related items selected from Table 1 that are usually taught in an introductory programming class. The items are the explicit learning activity from the reports mentioned earlier that are guiding curricula in information systems and computer science. We acknowledge that the intensity and level of complexity will differ from discipline to discipline, from program to program, and even from faculty to faculty, but we have provided a detailed programming example for discussion purposes.

The second column of Table 2 is labeled Implicit and identifies example higher level concepts from information systems, computer science, business, and other disciplines that benefit from a student having an understanding of the topic. Essentially, these can be seen as topics which support other, more advanced topics found in an information system or computer science curriculum.

The Appropriation column is used to suggest lifelong skills and possible careers which benefit from the implicit learning of the topic – the concept provides a possible analogy, metaphor, or example of the appropriation of the topic in 'real life'. In this way, this column suggests the 'transfer' skill associated with life skills associated with Lifelong Learning (AACU, 2015). These suggestions are based on a career track matrix in the IS 2010 report (Topi et al, 2010, Figure 6, p. 26), reviews and analysis of the career skill surveys mentioned earlier, and skills found in basic job descriptions. The appropriations are discussed in more detail for each programming topic below.

Table 2: Life Skills from Programming Areas

Explicit Programming Content Area (1)(2)	**Major or Career (Implicit)**	**Life Skills (Appropriation)**
Conditional Processing Variables, conditional statements, Boolean logic, etc.	Business Analyst, Lawyer, Software Engineer, Market Researcher, Insurance Agent, Credit Analyst, Security Officer, Baseball Team Manager, College Basketball Coach.	Tax preparation, Trouble-shooting electrical or mechanical systems, Balancing checkbook, Driving in rush hour traffic, Assembling a kid's toy, Cooking recipes.
Iterative processing Counted loops; conditional loops, searching, sorting, etc.	Quality Assurance Testers, Program Testers, Forensic Accountant, Logistics,	Lawn mowing, Painting a room, Vacuuming a rug,

	Musician, Actuaries, Banker, Assembly line worker, Bus Driver.	Waxing a car, Searching for lost keys, Training for track meet, Exercise routines.
Subroutines Functions, procedures, parameter passing, scope of control, etc.	Accountant, Mechanic, Architect, Mechanical Engineer, Systems Analyst, City Planner, General Manager, Wedding Planner, General Contractor, Nurse or Doctor.	Shopping lists, Planning a holiday meal, Cooking recipes, Doing math problems, Reading blueprints, Preparing taxes (worksheets, schedules, etc.). Reading contracts.
Human Computer Interaction GUI, Modular Design, Application Integration, Program Decomposition, I/O	Web Developer, Artist, Interior Designer, Salesperson, Game Designer, Kitchen Designer, Toymaker.	ATMs, Phone answering machines, Cash registers, Smart thermostats, TV cable box menu, Radio preset buttons.
(1)(Topi et al., 2010; p.56-57); (2) (ACM/IEEE, 2013; p.167-178)		

DETAILED PROGRAMMING EXAMPLES

Area 1: conditional processing - discounted price

Example: "You have the following variables: PURCHASE – the total of purchase by a customer; DISCOUNTED PRICE – the final price a customer pays. Given that the company provides a 10% discount on purchases over $1000, construct the IF – THEN – ELSE statement that provides a 10% of not. Store the result in DISCOUNTED PRICE."

The IF – THEN – ELSE structure is one of the three basic constructs in programming. It has at its base a Boolean condition and two sets of imperative actions. One set of actions is performed if the condition is true and the second set of actions is performed if the condition proves false. The concepts underlying this exercise include Boolean logic (single and compound conditions), conditional behavior, and logic flow. Each of these base concepts grow in complexity as students move from introductory level classes in business and information systems to the expectations of the upper division coursework.

```
If Purchase > 1000 Then
        Discounted_Price = Purchase * .90
Else
        ' no discount applied
        Discounted_Price = Purchase
End If
```

Often a business rule may require two independent conditions have to be true before an action is taken. This adds a level of complexity that can be solved in multiple ways. One way is with a compound condition that checks both situations in a single condition. The second method nests a second IF-THEN-ELSE statement on either the true or the false side of the first IF-THEN-ELSE. By adding a simple three word phrase into the business rule, the logic takes on a higher level of complexity.

Example: 'You have the following variables: PURCHASE – the total of purchase by a customer; DISCOUNTED PRICE – the final price a customer pays; RETURNING – a Boolean variable representing if the customer is a returning customer or a new customer. Given that the company provides a 10% discount *to returning customers* on purchases over $1000, construct the IF – THEN – ELSE statement that provides a 10% of not. Store the result in DISCOUNTED PRICE'.

```
If (Purchase > 1000) AND (Returning = "R") Then
        Discounted_Price = Purchase * .90
Else
        ' no discount applied
        Discounted_Price = Purchase
End If
```

The above situation has two independent conditions where the customer is 'returning' or not (returning customers have an 'R' in the variable), and the purchase price is over $1000 or not. The business rule wishes to reward only those situations where the customer is returning and the purchase is above $1000. The fact that the discount is only applied if both conditions are true allows for the use of the 'AND' compound condition. In essence there is still a binary decision in play – the purchase must be over $1000 and be made by a returning customer to qualify for the discount. Alternately, there is no discount.

However, the business may wish to seek out new customers and provide non-returning customers a 5% discount. At this point there are three potential outcomes (a 10% discount; a 5% discount; and no discount) and since a binary decision can only handle two branches, a second binary decision much be used.

Example: 'You have the following variables: PURCHASE – the total of purchase by a customer; DISCOUNTED PRICE – the final price a customer pays; RETURNING – a Boolean variable representing if the customer is a returning customer or a new customer. Given that on purchases over $1000, the company provides a 10% discount to returning customers and a 5% discount to new customers, construct the NESTED IF statement that provides the appropriate discounted price. Store the result in DISCOUNTED PRICE'.

```
If (Purchase > 1000) Then
      If (Returning = "R")  Then
         Discounted_Price = Purchase * .90
      Else  ' If not returning, then this is a new customer
         Discounted_Price = Purchase * .95
      End if
Else
      ' no discount applied
      Discounted_Price = Purchase
End If
```

In the above example, the three possible outcomes (Over $1000 and New; Over $1000 and Returning; Not over $1000) must be allowed for. This can be done with two IF-THEN-ELSE statements, but one of the outcomes must be nested within the second statement.

The logic in designing which condition operates as the first condition does have some impact on the efficiency or the processing, but not the effectiveness. There are situations where the number of outcomes makes using the basic IF statement unwieldy. The collection of sales tax is just one such case. Here each state has a different rate and therefor a different amount to collect for each purchase amount.

Example: 'You have the following variables: PURCHASE – the total of purchase by a customer; DISCOUNTED PRICE – the final price a customer pays; RETURNING – a Boolean variable representing if the customer is a returning customer or a new customer. STATE CODE – a two character field representing the state for which you need to collect sales taxes (OH, KY, IN). SALES TAX – variable for the calculated sales tax to be collected from the sale. TOTAL PRICE – the total price of the purchase after discount and including sales tax. Given that on purchases over $1000, the company provides a 10% discount to returning customers and a 5% discount to new customers, construct the NESTED IF statement that provides the appropriate discounted price. Store the result in DISCOUNTED PRICE. *On all sales, you should calculate the tax rate based on the STATE CODE and add that tax to the DISCOUNTED PRICE to obtain the TOTAL PRICE*'.

```
If (Purchase > 1000) Then
      If (Returning = "R")  Then
         Discounted_Price = Purchase * .90
      Else  ' If not returning, then this is a new customer
         Discounted_Price = Purchase * .95
      End if
Else
      ' no discount applied
      Discounted_Price = Purchase
End If
Select Case STATE_CODE
      Case is = "OH"
          Sales_tax = Purchase * .0575
      Case is = "KY"
          Sales_tax = Purchase * .06
      Case is = "IN"
          Sales_tax = Purchase * .07
      Case Else
          Msg "error"
End Select
Total_Price = Discounted_Price + Sales_tax
```

The SELECT CASE structure was developed to minimize the coding complexity created by sets of intertwined conditions and IF-THEN-ELSE statements that would be necessary; essentially fifty IF THEN ELSEs for the example here.

Area 2: iterative processing - loops

The iteration construct is one of the major constructs taught in a beginning programming class. Conceptually, a loop is used when you have a set of actions you wish to repeat on a

set of data. The set of actions are repeated either 1. A set amount of times – counted loops or 2. Until a condition is met – conditional loops. In our example, we want to determine how much our discounting policy has provided in discounts. Conveniently, we know there were 50 sales made this month. Without the functionality of loops one would have to make 50 copies of the above code and have 50 different names for the purchase variable, making for a long program. Loops usually work in conjunction with a type of storage which could be in computer memory or in a file on a disk. One analogy that is often used is processing a set of files in a file cabinet drawer. One takes out the first file, processes it, and then move to the next file. Loops work the same way. For purposes of our demonstration, we use the READ command to tell the computer to look at the next purchase, returning code, and state code.

Example: 'You have the following variables: COUNTER – a variable used to track the number of records; TOTAL DISCOUNTS – the accumulated total of discounts provided. PURCHASE – the total of purchase by a customer; DISCOUNTED PRICE – the final price a customer pays; RETURNING – a Boolean variable representing if the customer is a returning customer or a new customer. STATE CODE – a two character field representing the state for which you need to collect sales taxes (OH, KY, IN). SALES TAX – variable for the calculated sales tax to be collected from the sale. TOTAL PRICE – the total price of the purchase after discount and including sales tax. Given that on purchases over $1000, the company provides a 10% discount to returning customers and a 5% discount to new customers, construct the NESTED IF statement that provides the appropriate discounted price. Store the result in DISCOUNTED PRICE. On all sales, you should calculate the tax rate based on the STATE CODE and add that tax to the DISCOUNTED PRICE to obtain the TOTAL PRICE. There were 50 sales made this month. Calculate the total of all discounts provided.

```
For Counter = 1 to 50
  Read (Purchase, Returning, State_Code)
  If (Purchase)> 1000) Then
      If (Returning = "R")  Then
         Discounted_Price = Purchase * .90
      Else  ' If not returning, then this is a new customer
         Discounted_Price = Purchase * .95
      End if
  Else
      ' no discount applied
      Discounted_Price = Purchase ]
  End If
  Total_Discounts = Total_Discounts +
                   (Purchase  - Discounted_Price)
  Select Case State_Code
      Case is = "OH"
          Sales_tax = Purchase  * .0575
      Case is = "KY"
          Sales_tax = Purchase * .06
      Case is = "IN"
          Sales_tax = Purchase  * .07
```

```
      Case Else
          Msg "error"
  End Select
  Total_Price = Discounted_Price + Sales_tax
Next Counter
```

As alluded to above, we conveniently knew that there were 50 transactions. This allowed us to use the FOR … NEXT counted loop. Often we do not know how many times we must execute our look. For this we move to a conditional loop – one that ends when a text condition has been met. In the situation below we have a file of the fifty transactions, one per line, and we will process the transactions while there are still line items left in the file. The operating system manages a Boolean variable EOF as true if there are no more records or false if there are more records. The statement: 'There were 50 sales made this month' is amended to All transactions for the month have been put in a file SALES.DAT with one sale per line.

```
While NOT (EOF)
  Read (Purchase, Returning, State_Code)
…
…
…
Wend
```

The iteration processing here represents a simple example of the concept. More complex instantiations can process very complex situations. For example, loops within loops can produce very efficient and effective searching and sorting algorithms.

Area 3: subroutines - functions

Whether called procedures, subroutines, subprograms, functions, methods, etc., the main ideas communicated in these programming units are around 1. Clustering procedural logic that deals with the same objective or goal; and 2. Designing re-useable, callable code. Extracting one more example from our ongoing program, let's assume that we have found several more programs that must calculate the sales tax on a sales amount and the state in which the sale was made. We can create a function which contains the logic to calculate the appropriate sales tax amount based upon two pieces of information: the amount and the state.

```
Function SalesTax (Amount as single; StCode as string) as single
  Select Case StCode
      Case is = "OH"
          SalesTax = Amount * .0575
      Case is = "KY"
          SalesTax = Amount * .06
      Case is = "IN"
          SalesTax = Amount * .07
      Case Else
          Msg "error"
```

```
  End Select
End Function
```

This function can then be used in all of the programs where needed to get the sales tax on a sale by letting the function know the amount and the state. One of the teaching opportunities is to point out how the efficiency and effectiveness of the function grows; efficiency based upon the number of times the function is used and effectiveness based upon the number of states coded.

```
Total_Price = Discounted_Price + SalesTax (Purchase, State_Code)
```

Area 4: human computer interaction

The requirement to develop a good, workable, user interface for a computer program finds its ancestry in the study of human factors attributed to originating from WWII (Sanders and McCormick, 1987). Human factors design usually relates to physical machines or hardware and how they are operated by humans.

The discipline of human computer interaction (HCI) emerges from the prolific growth and use of computing in computers but also in (hardware) embedded systems. The evolutions of elevators, cars, ladders, lights, furniture, tools, etc. all fall inside the scope of a human factors or ergonomics discussion. Information technology tools emphasizing human factors would include hardware such as, easy open computer cases, keyboards, mouse, trackballs, video screens, desktops, laptops, tablets, etc. (Hewett et al., 2009). The challenge for the software developers is to take advantage of the new hardware updates.

However, the interaction that occurs between the user and the computer is different; essentially it is a conversation. In this conversation, the user and the computer converse through software using controlled protocols and language - in the general communication sense, not the programming language sense. These conversations are much more complex in that the interactions can become much more dynamic as both the computer and the human may get confused, require clarification, or not comprehend what the other needs or wants. Examples of HCI caused by embedded systems would include such interactions as; setting a VCR clock, touchpad conversation with computer systems on the ATM, cash register, phone, microwave cooking, etc.

How the computer talks to the user is important. For example, the assignment to develop an interactive pizza ordering program (figure 1) can create an active learning environment to; talk about the difference between mutually exclusive choices (radio buttons) and independent binary, cumulative, choices (check boxes) and the corresponding decision logic (IF THEN ELSE; Select Case) necessary to process the choices; appropriately display information; build interactive screen design; describe data validity issues; create unambiguous error message policies; etc.

Example: When the order button is pressed, the program code should identify which ingredients, what size and what type pizza is ordered. Using the costs from the menu below, determine the final cost of the pizza and display the total. You may use your own discretion as to the design of the code's logic (SELECT CASE; Nested IFs; Loops; Subroutines), but you must develop the interface to be a 'friendly' as possible. One example is in Figure 1.

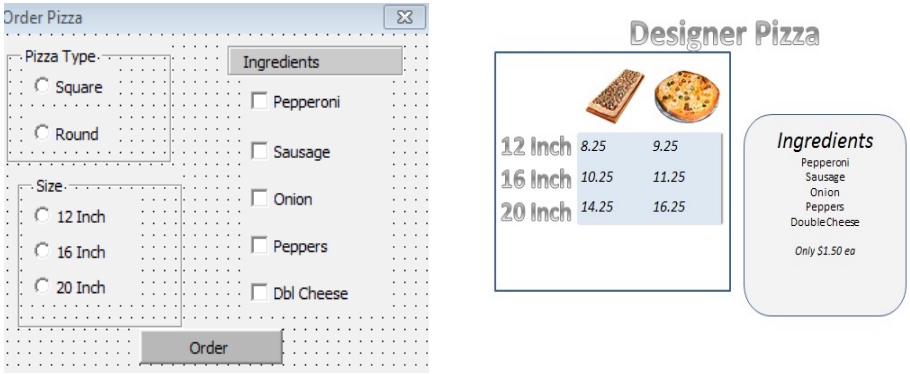

Figure 1: Example of Human Computer Interface

DISCUSSION AND CONCLUSION

Most people immediately see how the basic IF-THEN-ELSE statement applies across many activities. Constructing the appropriate conditional statement and building the imperatives are fundamental to decision making and problem solving. Understanding the nuances between using compound conditions, nested IF statements and SELECT CASE statements also provide core problem solving concepts. For example, these structures can help students later when they are required to design well-structured database searches or develop and apply security rights and authorizations. The logic flow inherent in nested IF statements provide a base foundation for discussions on project management topics. Clearly, the concept of organizing prerequisites and PERT/CPM charts fit well here. Probably the most direct use of the conditional action (IF-THEN-ELSE, NESTED IF, SELECT CASE) and the Boolean logic of compound conditions can be found in the decision making processes of advanced business courses such as Expected Value Tables, Analytics, Project Management, Marketing Research, Business Strategy, etc.

The application of the IF-THEN-ELSE construct permeates the development of life skills. Buying a car, filing taxes, deciding on what to order for lunch all boil down to identifying and understanding consequences if the a condition is true or false. The compound condition with an “AND” has the ability to help limit and refine choices; a car that is blue AND has 4 doors is more precise than a car that is blue OR has four doors.

The thought processes learned while clustering code and processing logic into contained sets of callable code – procedures - can be applied directly to other areas. Two examples would be clustering user requirements to make software development more efficient and the clustering of delivery locations to make delivery logistics more efficient. Those that can design well thought out procedures will probably be able to effectively cluster workflow activities together in project management tasks. When project managers clustering project activities into well-defined subprojects so as to more effectively and efficiently manage the larger project they have implemented the same underlying purpose of procedures, subroutines, functions, and objects with respect to the programming.

Remember, a function is distinguished as a set of code that concentrates on performing a specific role or activity. The inner logic of the subroutine code is not always open to inspection. As with the SalesTax example above, it usually accepts input parameters and provides output results. This characteristic can be found in a multitude of appropriations. One of the more obvious transfers of learning associated with the function (and procedures) is that of black box testing. The issues around testing functions and procedures would apply directly to designing test data for COTS (Customizable off the Shelf). Understanding how to vary input parameters and validate the corresponding output results could be a very valuable skill for accountants as they evaluate the candidates for an Accounting Information System (AIS).

The characteristics of functions and procedures also lend themselves to examples in life skills. One would be the lists one puts together for a day of shopping. One list identifies the stores you plan to visit; grocery, laundry, hardware, etc. and each store has its own list of items (bread, milk, cereal, etc.; pants, shirts, suits, etc.; hammer, hanging hooks, plaster, etc.,) respectively. Viewed in this manner, the store lists are functions called by the main 'to do' list. Each store list has a set of items and activities related to that store. Logically, putting them together makes a day long shopping experience more efficient and effective.

The inclusion of interactive programming in an introductory programming class opens a wide world of possibilities for analogies and metaphors. Everyone has experiences with embedded systems along with complaints and suggestions on how they would do things differently. With the proliferation of smart phones, tablets, car dashboards, etc. the reference disciplines are close to endless. The Association of Computing Machinery (Hewett et al., 2009) recognizes the breadth of HCI as it lists the breadth of supporting disciplines contributing to the discipline of human-computer interaction. These include but are not limited to: cognitive psychology, decision making; programming languages; data visualization; user satisfaction; technology acceptance and use, etc.

The design and implementation of a good, workable, user interface for a computer program becomes a wonderful exercise to educate students on key issues in how humans and computers interact. This activity presents an excellent opportunity to acquaint students with techniques and tools for issues that will later surface in core classes of a IS curriculum such as decision making, writing case analyses, developing user requirements, etc.

The application of the core HCI design principles are applicable life skills. Creating a good conversation means understanding how both sides think. This skill is applicable to many careers and life situations; buying a house, negotiating a contract, etc. One of the more important abilities acquired in this activity would be a higher capacity to better understand why things (embedded systems) don't work. Again, like with all the appropriations mentioned here, this one reverts back to the core problem solving skill mentioned as deficient in all of the skills gaps in the opening.

In conclusion, the paper sets forth the proposition that embedded within the ideas and concepts learned in a programming class hides the introduction to many of the foundational and life skills students need to complete their degrees and to succeed in careers. This suggests one should become a generalist rather than a specialist in the way one presents examples in a programming class. Expressly, the teaching tip is to include many different examples from everyday life within the problems presented for the students

to solve. Finding examples from reference disciplines and at multiple levels of difficulty and of association will prove invaluable as students develop their analogies and appropriations. The beginning programming class that acts as the reference course for this paper, has students from many disciplines that are required to take the class; engineers, library science, business, computer science, healthcare, accounting, and math education students just to name a few. All of these students can benefit from a programming class as they "appropriate" and apply the programming concepts to their own discipline. The challenge is to help these students as much as possible by providing them with the proper tools to succeed first in their own chosen careers and then in life as they develop skills of lifelong learners.

REFERENCES

AACU (2015). Association of American Colleges & Universities. Foundations and Skills for Lifelong Learning VALUE Rubric. http://aacu.org/value/rubrics/lifelong-learning accessed March 17, 2015

AAHE (1996). "What Research Says About Improving Undergraduate Education," *AAHE Bulletin*, April 1996.

ACM/IEEE (2013). Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science. Cooperative project of ACM, IEEE, and IEE Computer Society, 2013, http://dx.doi.org/10.1145/2534860

Anderson, C. and Gantz, J.F. (2103). SKILLS REQUIREMENTS FOR TOMORROW'S BEST JOBS Helping Educators Provide Students with Skills and Tools They Need, IDC, October 2013.

Barr, R. B. & Tagg. (1995). "From Teaching to Learning - A New Paradigm for Undergraduate Education," *Change*, Nov/Dec, 13-25, 1995.

Casner-Lotto, J, and Benner, M.W. (2006). Are They Really Ready To Work?. The Conference Board, Inc.,

Casserly, M. (2012). "The 10 Skills That Will Get you Hired in 2013". Forbes. Accessed March 4, 2015. http://www.forbes.com/sites/meghancasserly/2012/12/10/the-10-skills-that-will-get-you-a-job-in-2013/

Chickering, Arthur W. and Gamson. (1987). "Principles for Good Practice in Undergraduate Education," *The Wingspread Journal*, Johnson Foundation, Inc. June.

Doria, J., Rozanski, H.D., and Cohen, E. (2003). What Business Needs from Business Schools, *Strategy+Business*, Fall 2003, Issue 32. Accessed March 14, 2015 http://www.strategy-business.com/article/03305?pg=0

Fischer, K. (2013). A College Degree Sorts Job Applicants, but Employers Wish It Meant More, The Chronicle of Higher Education, vol 59, Iss. 26, March 8, 2013.

Goodman, M.J. Sands, A.M. and Coley, R.J. (2015). America's Skills Challenge. Educational Testing Service, Princeton, New Jersey. January.

Hall, D.T. and Mirvis (1995). The New Career Contract: Developing the Whole Person at Midlife and Beyond, Journal of Vocational Behavior, Vol. 47, Issue 3, pp 269-289.

Hewett; Baecker; Card; Carey; Gasen; Mantei; Perlman; Strong; Verplank. (2009)."ACM SIGCHI Curricula for Human-Computer Interaction". ACM SIGCHI. Retrieved March 6, 2015.

Martz, W. Benjamin, Jr. and Gabrielle Landof. (2000). "Information Systems Careers: A Comparison of Expectations, " *Journal of Computer Information Systems*, Vol. 40, No. 2, Winter 1999-2000, p41.

Minsky, M. (1988). *The Society of the Mind*. Simon and Schuster, 1988.

Newell, A. & Simon, H. (1972). Human Problem Solving. Englewood Cliffs, NJ: Prentice-Hall.

Overby, S. (2006). "How to Hook the Talent You Need," CIO, 2006, pp. 40-54, http://www.cio.com/article/24439/Staffing_How_to_Hook_the_Talent_You_Need, last accessed March 26, 2015.

Papert, S. (1980). *MindStorms*, Basic Books.

Piaget, J. (1929) *The Child's Conception of the World*, New York: Harcourt, Brace & Co, 1929.

Piateetsky, P. (2014). Most Demanded Data Science and Data Mining Skills. Accessed March 3, 2015. http://www.kdnuggets.com/2014/12/data-science-skills-most-demand.html

Roby, Thornton B. (1966). "Self-Enacting Response Sequences," *Psychological Reports,* 19, pp19-31.

Sandelands, L. E., Ashford, S. J., and Dutton, J.E. (1983). "Reconceptualizing the Overjustification Effect: A Template-Matching Approach," *Motivation and Emotion*, Vol. 7, No. 3, 1983

Sanders, M.S. and McCormick, E.J. (1987). *Human Factors in Engineering and Design.* Sixth Edition, New York: McGraw-Hill.

Schank, R. and Abelson, R. (1977). *Scripts, Plans, Goals and Understanding*, Lawrence Erlbaum.

Schank, R. (1988). *The Creative Attitude*, MacMillan.

Taylor, E. (2014). 11 Hottest Tech Jobs of 2014, US News Money, http://money.usnews.com/money/careers/slideshows/11-hottest-tech-jobs-of-2014/2

Topi, H., Valacich, J.S., Wright, R.T., Kaiser, K.M., Nunamaker, J.F., Sipior, J.C., and Vreede, G.J. de. (2008) "Revising the Undergraduate IS Model Curriculum: New Outcome Expectations," Communications of the Association for Information Systems (23:32) 2008, pp 591-602.

Topi, H., Valacich, J.S., Wright, R.T., Kaiser, K.M., Nunamaker, J.F., Sipior, J.C., and Vreede, G.J. de. (2010). "Curriculum Guidelines for Undergraduate Degree Programs in Information Systems," ACM/AIS 2010.

Thorndyke, P. and Hayes-Roth, B. (1979). "The Use of Schemata in the Acquisition and Transfer of Knowledge," *Cognitive Psychology*, Vol. 11, pp82-106.