Software Engineering as a Model of Understanding for Learning and Problem solving

J Paul Gibson
NUI Maynooth, Ireland
pgibson@cs.nuim.ie

Jackie O'Kelly
NUI Maynooth, Ireland
jokelly@cs.nuim.ie

ABSTRACT

This paper proposes a model which explains the process of learning about computation in terms of well-accepted software engineering concepts, and argues that our approach to understanding how problem-solving skills are acquired is an innovation over well-accepted learning theories and models. It examines how *all* students make sense of computational processes; by reporting on experimental observations that have been made with school children, and with university undergraduates. We observed little difference between children and adults with regard to how they learn about computation, and suggest that the strong similarities are due to a common set of problem-solving techniques which are fundamental to all problem based learning, in general, and learning about computation, in particular. To conclude, we demonstrate that our model — based on software engineering concepts — is useful when reasoning about the relationship between problem solving and learning to program.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: Computer science education; I.2.6 [**Learning**]: Misc.; D.2 [**Software Engineering**]: Misc.

General Terms

Algorithms, Experimentation, Human Factors, Theory

Keywords

Computing Education Research

1. INTRODUCTION

In the last few years, both authors have independently been involved in problem-based learning (PBL) experimentation, inspired by the successful work being done in teaching programming to computer science students (see [21], for example). The 1st author has primarily been interested in observing young children (in schools) as they play with games and puzzles, and solve problems. His goal is to try and get a better understanding of the *algorithmic learning* process in order to improve the way in which he teaches university students about software engineering. The 2nd author's motivation is primarily in using problem-based learning as a means of helping CS1 students to:

- think critically and be able to analyze and solve complex, real-world problems;
- work cooperatively in teams and small groups; and
- demonstrate versatile and effective communication skills, both oral and written.

The two authors — through discussion about their PBL observations — recognised a similarity between how children in schools solve problems and how young adults in university solve problems. This commonality was formalised by identifying software engineering techniques common to both groups of students; and they then validated — from comprehensive notes — that both groups had been observed applying these principles during the problem solving process.

This paper reports on the first steps towards constructing a theory that: *software engineering provides a good framework for reasoning about how children and adults learn to solve problems.* It motivates the construction of such a theory through critical analysis of already existing observations. A rigorous validation is far from being complete; but our own conclusisons are that such a theoretical model: merits further investigation, may provide insight into the fuzzy boundary between deep and shallow learning, and should help us to construct a *theory of problems* that would provide a foundation for reasoning about problem-based learning for software engineering and learning how to program.

1.1 Learning Theory and Models: our innovative approach

There are hundreds of well-published complementary, and competing, theories of learning. The highly cited review by Hilgard and Bower published over half a century ago[20] is a good introduction to the foundations of learning theory. In this paper, we focus on the well-accepted theories that have had most influence on our own research into the areas of cognition and problem solving, and teaching how to program: Piaget, Bruner, Guildford, Gardner, Papert, Schoenfeld and Bloom.

Cognitive structure is the concept central to Piaget's theory. (See the work by Brainerd[6] for a good overview and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICER'05, October 1–2, 2005, Seattle, Washington, USA.
Copyright 2005 ACM 1-59593-043-4/05/0010 ...$5.00.

analysis of Piaget's seminal contribution to the field.) These structures are used to identify patterns underlying certain acts of intelligence, and Paiget proposes that these correspond to stages of child development. Piaget's most interesting experiments, with respect to the work presented in this paper, focused on the development of mathematical and logical concepts. His theory has guided teaching practice and curriculum design in primary (elementary) schools in the last few decades. However, his work predates the development of software engineering as a discipline and it is therefore unsurprising that it does not make reference to any of the concepts within the model that we propose.

Piaget's theory is similar to other *constructivist* perspectives of learning (e.g., Bruner [7], which model learning as an active process in which learners construct new concepts upon their current knowledge and previous experience: information is selected and transformed, hypotheses put forward and tested, and decisions made. As a result of following this theory, teachers encourage students to discover principles by themselves: this is the foundation upon which problem-based learning is built. Our model is complementary to this work as the software engineering concepts make the process of learning much more concrete.

Similarites can be seen between the constructivist view and the *theories of intelligence* such as proposed by Guildford's *structure of intellect* (SI) theory [17] and Gardner's *multiple intelligences*[12]. Typically, these theories structure the learning space in terms of skills, for example: reasoning and problem-solving, memory operations, decision-making, and language. These skills are at a higher level of abstraction than those seen in the model that we propose.

Piaget's ideas also influenced the seminal work by Seymour Papert in the specific domain of computers and education[28]. Papert argues that children can understand concepts best when they are able to explain them algorithmicaly through writing computer programs. In his constructionist world, computer technology and programming play a critical role in helping children to learn. We support Papert's argument by demonstrating that programming (software engineering) skills may be fundamental building blocks of the learning process.

As computer scientists, we were also influenced by the domain of teaching mathematics. In particular, Alan Schoenfeld argues that understanding and teaching mathematics should be treated as problem-solving [30]. He identifies four skills that are needed to be successful in mathematics: proposition and procedural knowledge, strategies and techniques for problem resolution, decisions about when and what knoweldge and strategies to use, and a logical *world view* that motivates an individual's approach to solving a particular problem. Our *algorithmic understanding* view would suggest that the strategic decision making process is founded on fundamental software engineering principles and concepts.

To conlcude this brief review, Bloom's taxonomy[4] of educational objectives is a fundamental model of learning which provides a well-accepted foundation for research and development into the preparation of learning evaluation materials. It structures understanding into 6 distinct levels: Knowledge, Comprehension, Application, Analysis, Synthesis and Evaluation. An important aspect of Bloom's Taxonomy is that learning is incremental — each level must be "mastered" before progressing to the next. We do not believe that our model of learning will necessarily be incremental in the same way. It is for future research to analyse whether this is inconsistent with Bloom's model, and to examine the potential consequences for our model.

In this short overview of the learning theories that have most influenced our research — and most of the research in this area — it can be seen that we propose building a model of learning that is very different in nature to the generally accepted models; though not necessarily inconsistent with them. As software engineers and teachers of computer programming, we identified that our understanding of problem solving is at a different level of abstraction to any of the existing frameworks; yet it provides useful insights into the learning process, in general, and learning how to program (to solve problems), in particular. This was the main motivation behind our proposing the development of "a new theory of problems" that would be useful to computer science educators who are not necessarily experts in learning theory.

1.2 CS1: Learning in University

It is well accepted within the computer science community that first year students find programming difficult. There is an abundance of papers in the proceedings of SIGCSE[1] and ITiCSE[2] that confirm this. One of the major stumbling blocks for students is the abstraction of the problem to be solved from the exercise description[23]. In order to try to overcome this difficulty, the 2nd author introduced a workshop into the first year programming module[27, 22]. We used the term workshop to specifically emphasise a meeting of concerted activity, along with problem solving, among a small number of participants. The workshop involved the students working in groups to solve problems. Each student in the class was assigned to a formal group for an entire semester and each group was assigned a separate workspace. Attendance at the workshops was complusory.

It is our experience — and anecdotal evidence from lecturers in other universities concur with us — that once a student sits in front of a computer, they feel compelled to be using the keyboard or the mouse, and that they do not take the time to map out a solution to the given problem. Therefore, we deliberately prohibited the use of computers during the workshop. We gave the groups a basic framework (which they could change) to tackle the problem. This framework asked them to consider the problem as presented, clarify the kernel of the problem and the product to be developed and restate the problem in their own words. In addition, they could generate ideas/hypotheses through the use of brainstorming, identify the key issues and constraints, and develop a step-by-step set of instructions to solve the problem. The overall objective with the use of these workshops and peer learning groups was to:

- develop the students' problem solving skills,
- develop the students' critical thinking skills,
- encourage alternate approaches to problem solving through group work, and
- encourage deep learning approaches.

[1]ACM Special Interest Group on Computer Science Education — http://www.sigcse.org/

[2]Innovation and Technology in Computer Science Education — http://www.cs.utexas.edu/users/csed/iticse/

It was our belief that if we could improve the students' abilities in these areas there would be a discernible positive result. In a separate report we analyse the problems in validating such claims[26].

1.3 Refinement: Learning in Schools

The 1st author has, over a number of years, made a number of observations with respect to the way in which children (aged 3 to 18) learn to solve puzzles and play games. As a software engineer, he observed that children learnt through a process of refinement, and discovered and successfully employed well-known software engineering techniques. Even the youngest children exhibit "advanced" software engineering practices in the way in which they learn. In fact, it seemed strange that some of the software engineering techniques that university students find difficult to employ are a fundamental part of *algorithmic learning* (i.e. learning to think algorithmically — with the explicit need to make sense of computational processes — in order to solve problems).

In the schools, we run a series of sessions that revolve around the children solving problems using algorithms. The same problems are used with all students (irrespective of age); but, of course, the problems are open-ended in the sense that more advanced students are exposed to more advanced variations. Also, the time that each class spends with a problem may range from 30 minutes to a number of hours (spread over a number of sessions).

The following list gives a flavour of the types of problems that we have run in a number of schools: (1) Prime number recognition, (2) Searching, (3) Sorting, (4) Noughts and crosses (tic-tac-toe)[14], (5) The matches game, (6) Shortest Paths, (7)Maze Walking, (8) The 15-puzzle, (9) The Tower of Hanoi, (10) Magic Squares, (11) Scales weights, measures and balancing, etc. It is beyond the scope of this article to report on all these sessions. Rather, we report on 2 typical sessions – *The Tower of Hanoi* and *Guessing and Searching* – which illustrate the software engineering techniques we observed the children adopting as a natural part of learning.

It should be noted that the goal of the sessions in the schools is not to teach the children problem-solving skills. The goal is to observe the children when they are asked to solve problems. They will, of course, learn how to solve problems from being exposed to specific instances. Note also that we do not have specific learning objectives associated with specific problems. However, we do have specific things that we are looking out for (from previous observations). In the conclusions, we comment on the software applications that we have developed, and are developing, in order to make objective observations without the observer influencing the outcome of the sessions.

2. THE SOFTWARE ENGINEERING COGNITIVE TOOLS

In the following sections, all the examples are taken from simple software engineering exercises. The formal definitions we propose illustrate only that the concepts can be given precise meaning through formal methods. Our motivation is that these formal concepts in the domain of software engineering are ideal candidates for building a more scientific foundation for a vocabulary (i.e. a formal modelling language) when reasoning about problem solving, computational learning and *algorithmic understanding*.

The graphical notation, for non-deterministic finite state machines (NFSMs), that we use has formal semantics (see, for example [13]), but for the purposes of this paper our intuitive explanations should be enough to convey their meaning.

2.1 Refinement

Refinement is a relationship between an abstract specification and a more concrete specification, leading ultimately to something concrete enough to be executed[24, 2]. There are different ways in which refinement can be precisely defined, all of which are more or less intuitive. A constructive view is that refinement is a method of software construction that allows (abstract) specifications to be iteratively refined into other (more concrete) specifications resulting in an efficient implementation. Algorithm (process) refinement allows this translation to occur over a procedure or code fragment. Data refinement allows the data representation to be translated into a *better* data structure.

It should be noted that refinement is often conceptualized as removal of non-determinism. This view is particularly useful when analyzing games and puzzles where poor players resort to random moves in more or less challenging positions[3].

2.2 Example: Sorting and process refinement

Typically, an abstract specification of sorting a list of elements is as follows:

> **Input**: a list, I say, of n elements $I_0, \ldots, I_{n-1}$
>
> **Output**: an **ordered** list, O say, of n elements $O_0, \ldots, O_{n-1}$ such that:
>
> > O is a permutation of I, and
> > $\forall i, j, O_i \leq O_j \iff i \leq j$,

This can be refined into an algorithmic 'solution' to the problem of sorting:

1. Take the **Input** list I and copy into list O.
2. If O is **ordered** then return this as **Output**, else swap 2 randomly chosen elements of O.
3. Goto step 2

We can specify this algorithm as a non-deterministic finite state automaton (NFSA). In **figure 1**, we illustrate the notion of refinement when sorting a list of only 3 integer elements. The states are labelled with the list of integer elements. The transitions (all non-deterministic) are labelled by arcs and correspond to the random swaps. Note: we have not represented the null transitions (where an element is swapped by itself) in the NFSA. The terminating state — where the list elements are ordered — is shaded. This terminating state cannot be exited.

We can see that from any starting state, it is always possible that the terminating state will be reached. Some simple maths shows that the probability of terminating in the final ordered state tends to 1 as the number of random swaps tends to infinity. Temporal logic — through the notion of a *fair object* — could be used to specify that the system — implemented as an object — will always eventually terminate in an ordered state[16].

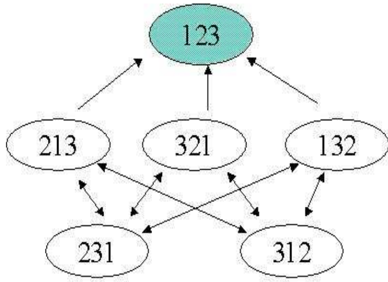

Figure 1: Random sorting of a 3-element list

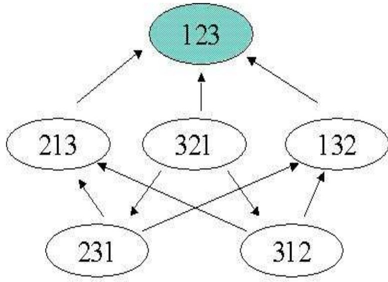

Figure 2: First sorting process refinement

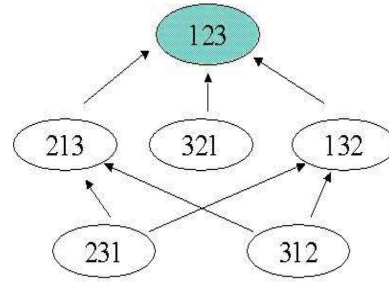

Figure 3: Second *optimal* sorting process refinement

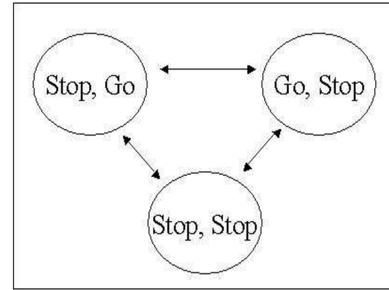

Figure 4: Traffic Light System

In effect, this NFSA represents a very inefficient solution to the problem of sorting a list of 3 integers. Refinements – that remove some of the non-determinism of the system (solution) – can be used to generate a more efficient solution. Consider the alternative solution in **figure 2**.

In this first refinement, we have removed all swap transitions that exchange elements that are already in order[3]. Now, simple reasoning – validated by a visual analysis of the unidirectional flow in the system — shows that the maximum number of transitions required to reach the terminating state is 3. The solution is better than the previous one because it is more efficient; where efficiency is measured by the number of fundamental operations — in this case, swaps — that must be executed in arriving at the final solution. The removal of non-determinism, in this case, has transformed a *correct* solution into a better *correct* solution.

To conclude this example, consider the (non-unique) *optimal* solution to this problem in **figure 3**, where further refinement of the original system has given an optimal NFSA.

2.3 Example: Traffic lights and *data refinement*

Consider the NFSA in **figure 4** that represents a traffic light system, where there are 2 lights for 2 flows of traffic.

In this case, the intention is that traffic can go through in at most one direction at any particular instance in time. Both streams of traffic can also stop at the same time (perhaps to let pedestrians cross the road).

Data refinement is concerned with adding structure to the way in which non-deterministic transitions occur, usually through the addition of new intermediate states. In this case, we may choose to have the data value `Stop,Stop` refined into 2 different values `Stop,Stop1` and `Stop,Stop2`, say. The NFSA on the left side of **Figure 5** shows how this is done.

In fact, this data refinement starts to make sense from the point of view of practical software engineering and problem solving when we combine it with an algorithm refinement. The system in the left side of **figure 5** is equivalent[4] to the original system in **figure 4**. A problem with both is that it is possible for the system to get into a cycle of states whereby one direction of traffic never gets to `Go`. In order to refine the behaviour of the system, we would like to ensure that such an *unfair* scenario is precluded. We do this by carrying out a process refinement — on the new system with 4 states — which removes the non-determinism in deciding which of the two `Stop,Stop` states are moved into when a `Stop,Go`

[3] We can also, in the same refinement, explicitly remove the null transitions where elements are swapped with themselves.

[4] We are using the notion of equivalence in an intuitive manner. There are many formal notions of equivalence which we could use to verify our statement.

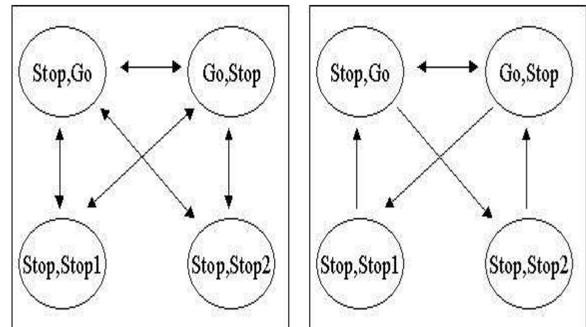

Figure 5: Traffic Light System - data refinement

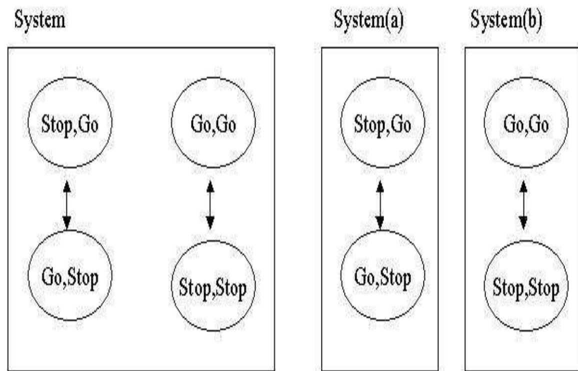

Figure 6: Traffic Light System - subclassing (specialisation)

or `Go,Stop` state is left. Then, from either one of the two `Stop,Stop` states we are guaranteed to return to a different state from which we have just come. This can be seen in the NFSA on the right hand side of **figure 5**.

2.4 Subclassing: Extension

In the traffic light example, imagine that we wish to have a means of recording the number of times that the system was requested by pedestrians (in state `Stop, Stop`). A natural software engineering solution to this problem would be to define a new traffic light system that carried an extra state counter. The behaviour of the system would be exactly as before, if we ignored the new count mechanism. In standard software engineering terminology, this is known as a subclass (extension) to the original system. (A formal treatment of this can be found as part of the classic *theory of objects*[1].)

Ignoring the extra count state variable reduces the NFSA back to the original behaviour. However, we have now extended its behaviour. The new system (subclass) does everything the old system (superclass) does, and more.

2.5 Subclassing: Specialisation

Another formally defined type of subclassing does not correspond to extending already existing behaviour, but intuitively corresponds to specialising it[25].

Consider the situation — illustrated in **figure 6** — where the behaviour of the system is strictly partitioned into 2 separate classes: (a) where the stop and go signals alternate and traffic is always flowing (in only 1 direction at a time); and (b) where the traffic is either completely stopped (in both directions) or flowing in both directions. There is no way for the system to move between these 2 separate sub-behaviours. We say that behaviours of system (a) and system (b) are specialisations of the original system.

2.6 Re-Use through Composition

In software engineering, software is usually not built from scratch. Normally, already existing software artefacts (from the set of documents and models that are built during the engineering of a software system - analysis, requirements, validation, design, verification, implementation, tests, maintenance, versioning and tools) are reused, in a wide range of ways, in the construction of a 'new' software system. Software reuse is one of most documented but least-well understood elements of the software engineering process[9, 11, 29, 10]. It would, of course, be surprising if we did not observe this type of re-use in the learning process.

2.7 Genericity: Universal and Constrained

In software engineering, modelling languages usually provide a means of specifying parameterised behaviour. Typically, we see this in the form of generic data structures. A classic example is that of a stack (with 'LIFO' behaviour offered through methods `push` and `pop`). This provides great opportunity for reuse: if you need a stack of integers and someone provides a generic stack then all you need to do is reuse their generic stack by instantiating the type parameter to integer. A stack is a classic example of universal genericity because it makes no assumptions about the types of things that are being pushed on and popped off.

In contrast to the universal genericity of a Stack, the first sorting example is a classic example of constrained genericity: we should be able to sort lists of any type of entity provided, of course, there is a way of comparing and ordering any 2 such entities. In effect, we are constrained to sorting only those things that can be sorted.

2.8 Re-using or re-usable

There is a trade-off in all software engineering between designing a system for maximizing re-use in the future; and for designing a system to maximize the re-usable entities from the past[8]. In the Traffic Light System example (see **figure 6**), there are two obvious types of re-use in play:

- it would be natural (for a software engineer) to transform the state into a composition of 2 boolean variables in order to be able to re-use an already existing, well-understood component; and

- it would be natural (for a software engineer) to make the traffic light system a re-usable component for re-use in larger, more complex systems.

The first type of re-use requires little additional effort on the part of the software engineer, and is a technique that is relatively easy to teach to students. The second type of re-use requires (typically) much more additional effort on the part of the software engineer, and is a technique that is relatively difficult to teach to students. Furthermore, in real industrial practice, there is a clear compromise between design for maximising re-use and design for maximising re-usability. We will propose, in our conclusions, that this trade-off may provide some insight into how to formulate the notions of *deep learning* and *shallow learning*, and how to distinguish between them.

3. THE CASE STUDIES

We chose to report on 2 studies: the first concentrates on observations in schools, the second focuses on observations with University students. Both these studies are representative of the wide range of problem case studies that we have executed and observed over a number of years.

3.1 Case Study 1 - Searching (in schools)

In the last 7 years, the following session has been run a total of 16 times with 16 different classes, at 10 different schools, with students as young as 6 and as old as 17. The mean age of the students having participated is 13. The

average class size is 18, with the minimum 10 and the maximum 35. In all cases, 1 teacher (or teaching assistant) is required to be present; and for above average class sizees we expect at least 2 such assistants. Our observations are from a mix of classes (and students).

As with all sessions, the searching sessions run in a sequence of phases. For each phase, interesting observations are highlighted and, where appropriate, we comment on aspects of learning where we believe that there is something fundamental in a model of algorithmic understanding (with respect to refinement, re-use and software engineering).

Given the large number of sessions that we have run, and the correspondingly large numbers of mostly informal observations that we have made, it is beyond the scope of this paper to analyse every interesting pattern or trend. However, we do identify some interesting aspects and provide informal analysis of these sample cases where appropriate. One of the goals of this paper is to stimulate thought as to what aspects merit more complete and more formal analysis.

Phase 1: observing pre-requisite understanding

When searching for an object from a collection of objects, we require only that a child can tell when 2 objects are the same. Through experience, we adopt a technique where sameness is based on some concrete property of the objects in question (size, shape, colour, texture, etc). In searching, we have had most success with lengths (the property) of pieces of string (the objects).

First we generate a reasonable number of pairs of pieces of string; where: the pairs are of equal length, and all pairs have different lengths. For example, a typical problem will have the following pairs, in no particular order, where the numbers represent lengths:

$$\{(4,4); (2,2); (5,5); (10,10); (16,16)\}.$$

Second, we separate the pairs into 2 collections:

$$\{4,2,5,10,16\}\text{and}\{4,2,5,10,16\}$$

Thirdly, we randomly mix each of them; giving, for example:

$$\{10,16,5,4,2\}\text{and}\{4,5,2,16,10\}$$

Finally, we ask the students to put them back into a collection of pairs. (This is similar to the problem of pairing socks, which many of them will be familiar with.) In the process of pairing, we confirm that all the children are able to compare the lengths of pieces of string and match those of equal length. The children do not need to actually solve the pairing problem in order to progress. Now that we know that children know how to check if 2 pieces of string have the same length, we can proceed to searching.In this case, we require only that a child can match a single piece of string with another piece of string in a collection. How you present the collection, and how you constrain their manipulation of the collection is key to observing the learning process.

We demonstrate that we can hide a piece of string in a box, and place a number of pieces of string in a number of boxes (one per box). Finally, we hand them a piece of string and ask them to find the matching string in one of the boxes. However, they are told that they can open only one box at a time; and that when a box is closed it must contain the piece of string that was in it originally. (With the youngest children it often takes a few minutes for them to understand the rules of the game.)

Through experience, children are much more enthusiastic and are more likely to actively participate in the sessions when there is an element of competition. In this case, we play the children against each other, playing alternate moves of the game. In this game, a move is looking in a box for the matching string. The first player to match the string wins the game. (Note that this does not require the younger students to be able to count.) All other children act as spectators of each game; and observing the spectators is as insightful as observing the players.

Phase 1 sample analysis: All students manage to play the game and show complete understanding of the rules.

Phase 2: first observations (process refinement)

We first observe the children selecting boxes in a purely haphazard, random manner. Although this, to begin with, is a game of chance, the children still seem to think that, for individual games, the winners are better players than the losers. The first interesting observation is when children realise that they have a better chance of winning if they never look in a box that they have already looked in. This observation usually arises from one (or more) of the children spectators shouting out that a player has already looked in a particular box and that they should choose another. In terms of software engineering, the children have quite naturally identified and communicated a process refinement (see **2.2**).

At this point, we ask children to play against each other using the new, improved approach. However, we preclude the spectators from speaking during a game. Very quickly, it is observed that some of the children have problems remembering in which boxes they have already looked (rather than having problems in following the new better search algorithm). We confirm this by increasing the number of boxes.

Phase 2 sample analysis: Approximately 50 percent of the youngest students (aged 6) are unable to play intelligently (through attempting to avoid boxes already examined). This percentage drops to zero as the students get older, so that all students aged above 9 are able to play with some intelligence (i.e. not purely randomly).

Phase 3: second observations (data refinement)

In the searching example, we observed 3 types of data refinement (see **2.3**) which the children introduced to overcome the problem of remembering which boxes had already been examined:

- Children searched the boxes in an ordered fashion (left to right, e.g.). In this way they had only to remember the last box searched in order to partition the collection of boxes into those already searched and those not yet examined.

- Children marked the boxes already searched (using a pen, e.g.). Now the partition was explicitly defined by each box having an associated boolean value (marked or not marked).

- Children moved the boxes from a “not yet examined pile” to an “already examined pile”.

Phase 3 sample analysis: The youngest student for which this type of data refinement was observed was 6 years old (who looked in the boxes from left to right). In all age groups there were a percentage of students who did not refine their game-play to help them remember where they looked; but this percentage dropped to almost 0 for the oldest students (where only 1 student failed to see the limitations of relying on their own memory).

Phase 4: ordering the data elements (constraining the problem in order to help find a better solution)

The next phase is to order the boxes based on the length of the strings within, before we ask the children to play the game. Very quickly it is observed that not all the children realise that the strings in the boxes are ordered by length. The children who realise the data are ordered are observed playing in a more structured manner. (Note that the notion of ordering in this case introduces a new pre-requisite: it is no longer sufficient for children to be able to check if two strings are of the same length, they now have to be able to tell the relative ordering of the strings.)

Over a period of time, we observe that the children effectively refine their solution[5] to a binary search where they do not always optimise the search by cutting the search space perfectly in two every time they make a guess. They know they need to look to the left or right of the current string box, based on the relative sizes of the search string and the string in the box.

With children older than eight years, we almost always observe at least one of the children adopting a "nearly perfect" binary search. They are observed trying to explain their *algorithm* to their colleagues. This form of refinement is fundamental to software engineering, is difficult for university students to understand[15] and apply, yet is observed in school children when they learn to play a game through competition.

The children are asked if it is possible to play better? Often they make quite solid arguments as to why their solution is optimal. At this stage, we play against them and we always find the string that is being looked for in the first guess. We are employing a perfect hashing function, based on knowledge of additional structure to the data values, to map the string length directly to a particular box. They often accuse us of cheating; and only the more advanced students realise the trick. (We do not explain it to them if they do not see it themselves; and often we are contacted by teachers and parents asking for us to explain the trick!)

Phase 4 sample analysis: We have witnessed one 6 year old boy attempting to explain their use of a binary search. This is an exceptional case, as the next youngest to discover binary search (that we have witnessed) was 8 years old. There appears to be a clear boundary at the age of 9 where nearly all the children appear to discover binary search; and through discussion with the teachers this seems to coincide with the introduction of fractions and ratios in their mathematics classes.

Phase 5: generalising the problem

After the children have managed to refine an algorithm for searching for a string of a certain length, we replace the strings by some other physical entities. In most instances, we use weights or balls (of different size). *All* children manage to generalise their solution for strings to other physical entities which can be ordered in some intuitive fashion. This is an example of constrained genericity (see **2.7**).

Phase 6: working with abstractions

I tell the school children that I am thinking of a number between two integer values (usually 0 and 100). The game is that they have to guess the number I am thinking of, by asking me questions to which I am allowed to answer only *yes* or *no*. Quite quickly, we observe that some of the children employ the same algorithm for the "guess the number game" as they do for the "find the string game". We observe that others do not.

Phase 6 sample analysis: The age at which 50 percent of the children (in that class/age group) can work with the number abstraction is between 11 and 12 years old.

Phase 7: observing compositional re-use and subclassing (specialisation)

I tell the children[6] that I am thinking of 2 numbers that add up to 100. I ask them to find both, following the same guessing framework as they have already seen with the strings and "guess the number" games. Before we start to play some games, I ask them to tell me whether this game is more/less difficult than the first (in terms of the number of moves that it will take to find the answer). Surprisingly, most children state that this 2-number game is more difficult because you have 2 numbers to find. In a similar vein, I ask the children to find a number that I am thinking of (between 0 and 100), but I also tell them that it is odd (or even). In this instance, they state that this game is easier than the original (even though the size of the search space is the same for each of these variations.)

For the minority of the children who think the 2-number game is easier, I ask that they explain their reasoning. Typically, they provide a constructive specification of how to play the variation using the mechanism that they have refined for playing the original game (the perfect binary search):

> "use the previous search to find the smallest number (which must be in the range 0 - 50), and then calculate the largest number as $(100-smallest)$."

Of course, this search will, on average, be 1 step quicker that the search for a single number in the range 1 — 100. Some of the children manage to explain this improvement. There are two components that are being re-used here (see **2.6**):

- the original binary search, and
- the calculation of the largest (missing number) as a simple subtraction.

A second noteworthy observation is that the children quickly identify a symmetry in the problem that means that they can reformulate their approach in terms of finding the largest (not smallest) element first. This symmetry can be viewed as a form of sub-classing (see **2.5**) where either approach

[5] We argue that this solution implies an implicit understanding of an algorithm.

[6] This phase requires us to work with older children who have the ability to count to 100, and perform simple addition and subtraction.

is functionally correct; but where one implementation may be more or less efficient than another under certain circumstances; in other words, they are each specialisations.

Phase 7 sample analysis: The youngest student to correctly identify that the 2-number game is easier than the 1-number game was 10 years old. The oldest student who did not understand the explanation of why this should be the case was 16 years old.

Final Phase: Communication Observation

The number of phases that we execute can vary between 5 and 15 (depending on the school and children). We normally terminate the session with a final phase that helps us to make more meaningful observations with respect to children having really achieved some sort of *algorithmic understanding* of the game they are playing. Where possible, we mix the children from the session with children who have not participated. We routinely observe the children who were involved in the session playing the games against their friends. Some of them keep the algorithmic secrets to themselves in order to improve their chances of winning, others take more delight in explaining the algorithms they have learned to their colleagues.

Final Phase — sample analysis: The percentage of students who attempted to teach the algorithms to their peers did not appear to be correlated with the age of the students. Similarly, we could not identify any pattern with respect to students keeping the secrets to themselves and their ages.

3.2 Case Study 2 - The Tower of Hanoi

The work described in this section of the paper was carried out over two years. It is based on our recorded observations of the interaction and communication between CS1 students within groups and each group's written notes and solutions to problems. For each student in every workshop we used a Likert type scale of 1 to 5 and rated their ability to support their beliefs, have effective communication skills, participate in the groups, be open to new ideas and show constructive critical thinking[22]. We also rated the overall performance of the group in how they tackled the given problem, if they achieved the learning outcomes, and if they were successful in reaching a solution to the problem. As each group were required to produce a written set of instructions on how to solve the problem and file a master copy in their group journals, we were able to analyse and compare each group's solution to a given problem. We also used peer-group formative assessment, whereby groups assessed each others work with the proviso that any criticism had to be constructive. In addition, we video recorded every group in one of the workshops. In all, there were thirty-eight formal groups over the two year period, with a group size of five to seven students on average. In their allocated workspace each group had the use of a white board and were facilitated by a post-graduate student, the 2nd author acted as a roaming facilitator visiting and observing all groups in every workshop. The 2nd author and the post-graduate students met each week to discuss the previous week's workshop and review each group's performance, the 2nd author recorded these meetings.

The Tower of Hanoi puzzle, invented in 1883 by Edouard Lucas [18] has been widely used as a programming exercise in introductory computer science data structures and algorithm courses[7]. It has also played an important role in experiments in the area of child psychology and learning difficulties (see [5] for a typical example). The traditional Tower of Hanoi consists of three pegs and a set of n, typically six to eight, disks of differing diameters that can be stacked on the pegs. The objective is to transfer the disks one at a time, from an initial start state into a goal end-state in the minimum number of moves, subject to the rule that a larger disk can never be placed on top of a smaller disk.

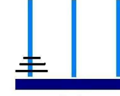

Figure 7: Prop in its starting position

This problem was given to the students in the workshops in the first week of term. During lecture time (prior to the workshops) in that week the students looked at different types of instructions and the language used to convey those instructions; for example: musical scores, knitting patterns, instructions for assembling an object and cooking recipes. We noted the precision given in some instructions and the ambiguity in others. We had not covered any (Java) programming syntax.

Our objectives for the use of this problem were that the group should:

- break the problem down into solvable units,
- identify the repeated sequence of moves required in solving the problem,
- label/name discs and pegs,
- use conditional statements, and
- recognise that a solution may not be the most efficient.

We gave a prop (see **figure 7**) to half of the groups. All groups identified the following:

- **Facts:** 3 pegs, n discs of decreasing size.
- **Constraints:** Move only one disc at a time, cannot place bigger disc on to a smaller disc.

Observations of the "no prop" group category

These groups spent a significant amount of time trying to decide how they would represent the pegs and discs. Some of the initial ideas were based around using chairs as the pegs and people as the discs, discussions ensued about how to represent the different size of discs: should it be the tallest person to the smallest person, or the heaviest person to the lightest person? Another approach was the use of coins to represent the discs and everyone was to assume that pegs existed; subsequently pegs were drawn on a piece of paper

[7]It is difficult to find a data structure and algorithms book that does not analyse the Tower of Hanoi problem; a typical introduction to the problem can be found in the classic *Walls and Mirrors* book [19]

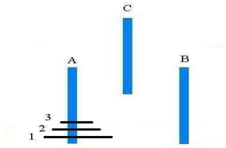

Figure 8: Re-designed prop in its starting position

and the coins were moved on the paper. Another idea was the use of books of different sizes as the discs and these were placed on a table, lines were drawn on the table to signify the pegs. Each group in this category devised some strategy for visualising the tower. All of the groups made use of the white board to document their moves. All groups put a name or label on the discs and the pegs. In terms of software engineering the students have used data refinement and abstraction (see **2.3**) when they:

- discussed different ways to represent pegs and discs;
- investigated the use of chairs, people and books; and
- put a label on the discs and the pegs.

We observed the use of constrained genericity (see **2.7**) in that were able to solve the problem using any type of entity (coins, people, books) provided there was a way of comparing these entities in terms of size (larger, smaller).

Observations of the "prop" group category.

These groups engaged in a very similar process as the groups with no props. However, the use of the prop meant that they were immediately able to visualise their ideas and test out their strategy at a much earlier stage. We noticed that the groups using the props did not use the white board and relied solely on the prop. All groups put a name or label on the discs and the pegs.

One of the groups in this category initially followed the same process as the other groups. What was interesting about this group was that they subsequently threw the prop away as they believed the prop was designed incorrectly. They identified a "solution" incorporating a circular pattern (see **figure 8**), where the moves were refined to be *clockwise* and *anti-clockwise*.

Find a better solution

Once a group had devised a solution for at least 3 and 4 discs, the observer intervened and asked them if they could solve the problem with the same constraints in less moves? This question was asked of the group irrespective of the solution that a group had formulated. This forced the groups to retrace their steps and to question whether this was the best solution they could devise for the instance of the problem. The groups in the "no prop" category had their sequence of moves recorded on the white board; and they counted the number of moves it had previously taken them and then tried to change their sequence to improve upon their solution. This involved experimentation with random moves and corresponded to them testing out sequences of process refinements (see **2.2**), similar to the random swapping being refined in our sorting example(see **figure 2**). The groups with the prop had difficulty remembering the exact sequence of moves they had made; it was at this stage that they made use of the white board. One student took instructions from the group to make the moves, a second student called the moves made, while another student recorded those moves on the white board.

A number of groups in both categories reduced the problem to 1 disc, came up with two different solutions, and selected the solution with the least moves. They then introduced another disc and went through the same process, once again selecting the solution with the least number of moves. This process continued until the students started to identify the same sequence of moves in their solution for 2 and 4 discs and also between 3 and 5 discs. They hypothesised that the sequence of steps for 2 and 4 would hold for 6 discs, subsequently tested this by methodically recording each move on the white board and confirmed their hypotheses held true.

The techniques used by the students here are:

- Process refinement (see **2.2**) — when they solved the same problem in less moves.
- Subclassing and genericity (see **2.5** and **2.7**) — when they identified a different sequence for odd and even number of discs.
- Composition (see **2.6**) — when they identified re-use for the repeat sequence of moves for 2, 4 and 6 discs.

An optimal solution

We noticed that once a group confirmed a sequence of moves was repeated they came up with a formula which would determine the optimal number of moves for solving the Tower of Hanoi with n disks. The majority of groups in the no prop category used formula (**1**).

(1) moves for $n =$ (moves for $n-1) * 2 + 1$

Whereas the groups who used the prop tended to develop formula (**2**).

(2) 2^n - 1

In both cases n represents the number of discs.

We have taken this classic problem and in the first week of term, without giving the students the computer science or software engineering background, asked them to solve it. They have demonstrated to us, through their process of solving the problem and in the concrete step-by-step set of instructions that they produced, their implicit use and understanding of software engineering techniques. It appears that this *algorithmic understanding* is made up of the same components as we observed in the school childrens' problem solving skills.

4. COMPARISON

We have already discussed the similarities between the two groups of students, based on the common types of software engineering techniques that we observed. For completeness, we comment on the key differences between working with these two groups.

4.1 Communication and objective validation

A main difference is that school children and university students have different communication skills. In order to

validate an observation that a subject has indeed some form of *algorithmic understanding* one must take into account the communication skills of the subjects. The two extremes of the spectrum are:

- **Very young children** who cannot write. They do not have a large vocabulary for speech. They can, however, communicate through a natural combination of speech, drawing, sounds and gestures. To validate that a young child has *algorithmic understanding* we have introduced a final phase to all our sessions. This phase mixes children who have just finished their game-playing session with children who have not participated. We observe that many of the children successfully communicate their playing *algorithms* to their friends.

- **University students** who have good communication skills (written and oral). To validate their *algorithmic understanding* we ask simply that they write down (in whatever language or notation they wish) what it is that they are doing.

In both cases, there are different strengths and weaknesses to this type of validation. An example of the material returned by the university students for the Tower of Hanoi problem (who had access to the prop) is given below:

```
Note that there are two different patterns,
one for odd and one for even.

Sample solution for 3 and 4 discs.

1: Count the number of discs
2: If number of discs on peg 1 is odd,
            move top disc (n) to peg 3
   if even move top disc (n) to peg 2 go to step 10
3: Move n+1 to free space
4: Move n to n+1
5: Move n+2 to free space
6: Move n to peg 1
7: Move n+1 to n+2
8: Move n to n+2
9: Goto 24
10:Move n+1 to free space
11:Move n to n+1
12:Move n+2 to free space
13:Move n to n+3
14:Move n+1 to n+2
15:Move n to n+1
16:Move n+3 to free space
17:Move n to n+3
18:Move n+1 to free space
19:Move n to n+1
20:Move n+2 to n+3
21:Move n to free space
22:Move n+1 to n+2
23:Move n to n+1
24:No more discs STOP
```

It is beyond the scope of this paper to analyse the degree of evidence that is required in order to state that a subject has achieved *algorithmic understanding* of a problem or game. However, in the example above, such understanding has — we argue — been clearly demonstrated.

4.2 Size of groups and time with classes

We note that there are 2 major differences with respect to the organisation of the sessions:

- **Group size**: In the school classroom, we do not run parallel sessions across a number of groups. Thus, our group size is effectively the number of children in the class. In the university, we explicitly divide the students into small groups.

- **Time for sessions**: In the university, each session is strictly controlled to last a set period of time (usually 90 minutes) in a single period. In the schools, we run sessions over a variable period of time with multiple visits to cover a sequence of phases. Thus, the amount of time spent on a problem or game is not fixed. The shortest sessions are 30 minutes. The longest totalled 4 hours (over 3 visits)

It is beyond the scope of this work to examine how the organisational differences could impact on our experimentation and observational analysis.

4.3 Maturity of subjects and deliberate "misbehaviour"

We have already commented on the impact of age on the ability of the subjects to communicate their *algorithmic understanding*. In schools we have observed behaviour that never arises (as far as we have observed) in university; namely, subjects deliberately miscommunicating their understanding. It is not uncommon for us to observe a child playing the search game using a near perfect binary search. However, when asked to explain what they are doing, they deny any knowledge of their structured approach and then simulate random playing when they know we are watching. There are a number of explanations (usually supported by their teachers) for this type of behaviour which are consistent with our observations. We are moving towards a more scientific objective collection of data using computer applications to simulate the game and collect data in an unobtrusive way: this should overcome the problem of deliberate "misbehaviour".

5. CONCLUSIONS

We believe that the work presented in this paper motivates further investigation and experimentation. We observed little difference between the children and adults with regard to how they learn about computation, and suggest that the strong similarities are due to a common set of problem-solving techniques which are fundamental to all problem-based learning, in general, and learning about computation, in particular. This common framework is usefully modelled using well-understood software engineering techniques.

Future work falls into 3 complementary streams:

- Develop a *theory of problems* that could be used in reasoning about the order in which problems should be presented to CS1 students, and the relationships between these problems.

- Implement a collection of Java applications for objectively gathering data about the way in which school children solve problems. Some prototypes — for sorting and searching games — have already been successfully deployed[8].

[8]Thanks to Marie Nangle for her final year project *Algorithms for Understanding Algorithms — an interactive tool for analysis of the way in which children learn algorithmic concepts* in 2002, and Pat Phelan for his MSc Software Engineering thesis *Rapid prototyping an educational online game: experimenting with sorting* in 2004.

- Feed back what we have learned from this collaborative research into the sessions that we run in the schools and with the university students.

Our final objective is to consolidate our proposal for the theory that: *software engineering provides a good framework for reasoning about how children and adults learn to solve problems.* This paper is a first step towards that goal.

6. ACKNOWLEDGMENTS

Both authors would like to thank the school children (and schools) and students who participated in the sessions. We also thank the referees of the paper for their comments.

7. REFERENCES

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[2] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[3] R. J. R. Back and J. von Wright. Contracts, games and refinement. *Theoretical Computer Science*, 230(1-2):259, 2000.

[4] B. S. Bloom and D. R. Krathowl. *Taxonomy of educational objectives*. McKay & Co, 1956.

[5] S. Borys, H. H. Spitz, and B. A. Dorans. Tower of Hanoi performance of retarded young adults and nonretarded children as a function of solution length and goal state. *Experimental Child Psychology*, 33(1):87–110, 1982.

[6] C. Brainerd. *Piaget's Theory of Intelligence*. Prentice-Hall, Englewood Cliffs, NJ, 1978.

[7] J. S. Bruner. *Toward a theory of instruction*. Belknap Press of Harvard University, Cambridge, Mass, 1966.

[8] W. Frakes and C. Terry. Software reuse: metrics and models. *ACM Computing Surveys*, 28(2):415–435, 1996.

[9] W. B. Frakes and S. Isoda. Success factors of systematic reuse. *IEEE Software*, 11(5):14–19, 1994.

[10] C. Gacek, editor. *Software Reuse: Methods, Techniques, and Tools, 7th International Conference, ICSR-7*. Lecture Notes in Computer Science, Vol. 2319, 2002.

[11] G. C. Gannod and B. H. Cheng. A framework for classifying and comparing software reverse engineering and design recovery techniques. In *6th Working Conference on Reverse Engineering*, pages 77–89. ACM Int. Conf. Series, 1999.

[12] H. Gardner. *Frames of mind: the theory of multiple intelligences*. Basic Books, New York, 1983.

[13] J. P. Gibson. *Formal Object Oriented Development of Software Systems Using LOTOS*. Thesis csm-114, Stirling University, Aug. 1993.

[14] J. P. Gibson. A noughts and crosses java applet to teach programming to primary school children. In *PPPJ '03: Proceedings of the 2nd international conference on Principles and practice of programming in Java*, pages 85–88, New York, NY, USA, 2003. Computer Science Press, Inc.

[15] J. P. Gibson and D. Méry. Teaching formal methods: Lessons to be learned. In *2nd Irish Workshop on Formal Methods*, Cork, Ireland, July 1998.

[16] P. Gibson and D. Méry. Fair objects. *Object-oriented technology and computing systems re-engineering*, pages 122–140, 1999.

[17] J. P. Guilford. *The Nature of Human Intelligence*. McGraw-Hill, New York, 1967.

[18] D. Harkin. On the mathematical works of Francois Edouard Anatole Lucas. *Enseignement mathematique*, 3:276–288, 1957.

[19] P. Helman and R. Veroff. *Intermediate Problem Solving and Data Structures: Walls and Mirrors*. Benjamin Cummings Publishing Company, Menlo Park, California, 1986.

[20] E. R. Hilgard and G. H. Bower. *Theories of learning*. Appleton-Century-Crofts, New York, 1956.

[21] J. Kay, M. Barg, A. Fekete, T. Greening, O. Hollands, J. H. Kingston, and K. Crawford. Problem-based learning for foundation computer science courses. *Computer Science Education*, 10(2):109–128, august 2000.

[22] J. O. Kelly, S. Bergin, S. Dunne, P. Gaughran, J. Ghent, and A. Mooney. Initial findings on the impact of an alternative approach to problem based learning in computer science. In *Problem-Based Learning International. Conference 2004: Pleasure by Learning*, July 2004.

[23] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In *ITiCSE-WGR '01: Working group reports from ITiCSE on Innovation and technology in computer science education*, pages 125–180, New York, NY, USA, 2001. ACM Press.

[24] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, 1987.

[25] E. Nantajeewarawat and V. Wuwongse. Nonmonotonic inheritance through specialisation. In *DOOD '97: Proceedings of the 5th International Conference on Deductive and Object-Oriented Databases*, pages 423–424, London, UK, 1997. Springer-Verlag.

[26] J. O'Kelly and J. P. Gibson. Pbl: year one analysis - interpretation and validation. In *Problem Based Learning 2005*, 2005.

[27] J. OKelly, A. Mooney, J. Ghent, P. Gaughran, S. Dunne, and S. Bergin. An overview of the integration of problem based learning into an existing computer science programming module. In *Problem-Based Learning International. Conference 2004: Pleasure by Learning*, July 2004.

[28] S. Papert and J. Sculley. *Mindstorms: children, computers, and powerful ideas*. Basic Books, New York, 1980.

[29] R. Rada. *Software Reuse*. Intellect Books, 1986.

[30] A. H. Schoenfeld. *Mathematical Problem Solving*. Academic Press, Orlando, Fla, 1985.