

# Patterns in Learning to Program - An Experiment?

Ron Porter<sup>1</sup> and Paul Calder<sup>1</sup>

<sup>1</sup> School of Informatics and Engineering  
Flinders University of South Australia,  
PO Box 2100, Adelaide, South Australia 5001,  
Email: ron.porter, calder@infoeng.flinders.edu.au

## Abstract

Learning to program involves the application of programming language features to the solving of novel problems, and the experience of educators suggests that it is this factor that causes novice programmers the most difficulty. Because software patterns are descriptions of common problems and their solution written in a standardised format that facilitates reuse, their use in the novice context is indicated. An earlier paper (Porter & Calder 2003) suggested and demonstrated a process for applying patterns to problems that derives from the relationships between patterns in a pattern language suitable for novice programmers. This paper reports on the feasibility of testing the idea.

*Keywords:* design patterns, programming, novice.

## 1 Introduction

Design patterns were first introduced into the Computer Science domain in the mid 1990s. The patterns introduced at that time were high level programming concepts and the pattern idea has been widely adopted at the advanced programming level. The real power of the pattern concept derives from their incorporation into the structure that is called a pattern language. (Alexander 1979, p.324)

It is this structure that supplies the information about the context of each pattern and enables a sequence of patterns to be put together. This is important because most problems are not simple enough to be dealt with in a single pattern. The pattern language provides the process by which patterns that solve part of a larger problem can be put together in the same way that natural language allows small ideas to be put together in order to express larger ideas.

In an earlier paper (Porter & Calder 2003) we suggested that there is a process for applying patterns to problems that derives from the relationships between patterns in a pattern language. This idea is discussed fully in the earlier paper with a demonstration of the process in a simple programming problem. The patterns used in the process are mainly based on programming language features, but the critical factor in their use is their incorporation into a pattern language structure from which the design process derives. As the main problem for most novice programmers lies in the application of the programming language constructs to the solving of novel programming problems

the idea of using this pattern process to assist novice programmers arises.

The idea of using patterns in introductory programming is not new. Joe Bergin (Bergin n.d.), Eugene Wallingford (Wallingford 2001), and many others have presented patterns for this field. Mostly, however, the language aspect of Alexander's concept has not been fully developed and it is the language that provides the generative power of the pattern idea. (Alexander 1979, p.315) Indeed Alexander himself, at the OOPSLA conference in 1996, pointed out that the adoption of his thinking by the Computer Science community has been incomplete. He said that he could see lots of patterns, but no generative structural order - no process or language.

In helping novices to program, it is this generative aspect of the whole pattern language that is important, rather than just the patterns themselves. The rest of this paper is a discussion of the proposal to test the use of patterns, presented in pattern language form, in the novice programmer situation. Because designing experiments to test the skills of people is notoriously difficult, we decided to run a pilot study to test the design of the experiment. We hoped that the trial run would also enable us to assess the degree of involvement required of the participants and point to the factors that we would need to consider in ensuring a successful experiment.

## 2 The Pedagogy of Teaching Programming

### 2.1 The Problem of Teaching Programming

Because it requires the learning of a skill, most educators involved in teaching programming agree that many students struggle in this field.

Results from a recent project by McCracken et al. (2001) are compelling, because of the number of authors from differing educational institutions and cultures. The 10 authors teach introductory programming across 8 universities, in 5 countries. Each author tested his/her own students on a common set of programming tasks. The students performed much more poorly than the authors had expected. The students did not simply fail to complete the set task, most students did not even get close to solving the task.

(Lister & Leaney 2003)

There are many studies into the relationship between programming and problem solving ability. (Mayer, Dyck & Vilberg 1986)(VanLengen & Maddux 1990)(Reed 1998)(East & Wallingford 1997)(Haverty, Koedinger, Klahr & Alibali n.d.) These concerns have

tended to flow into the idea of using patterns in introductory programming because of the resonance of the apparent cognitive difficulties exhibited by novices in building solutions with the problem-solution pair aspect of patterns.

Much psychological research ..... suggests that programming expertise is partly represented by a knowledge base of pattern-like chunks, variously named plans, templates, schemas, or idioms. .... Components of such a chunk resemble those described in Design Patterns. ....Additional research suggests that students gain expertise in programming and other disciplines from a process of knowledge integration.

(Clancy & Linn 1999)

The suggestion, here, is that patterns are useful in the integration of knowledge, in the task of putting disparate concepts and ideas into a coherent form, and this should be advantageous to novices.

However there is only one way to rigorously test a new approach to a learning problem, and that is to run an experiment based on a comparison of the new with the old. Qualitative assessments of effectiveness are easier and quicker, but it is difficult to be definitive about the results.

Given the current conditions, it is especially important to distinguish truth from assumption, to have practice that is well-founded. Evolving teaching practice is normal to good teaching, but evaluation reliant on anecdote is not good enough. Adding a research perspective allows educators to learn more from their practice.

(Daniels & Berglund 1998)

## 2.2 Bloom's Taxonomy

Considering a place for patterns in the learning process requires an attempt to understand the learning process. In the late 1940s, work began on the task of classifying education goals and objectives. The work on the cognitive domain is commonly referred to as Bloom's Taxonomy of the Cognitive Domain.(Bloom 1971) The idea of the taxonomy is that educational objectives can be arranged in a hierarchy from less to more complex. Table 1 shows how Bloom's taxonomy relates to some of the key tasks in programming.

Bloom's categories	Learning to program
Knowledge	Tools, constructs, syntax
Comprehension	Relating concepts
Application	Flow, semantics
Analysis	Understand the problem
Synthesis	Create the solution
Evaluation	Assess other options

Table 1: Bloom's Categories in Programming Terms

Using the term in the general sense, patterns are important at all levels in the cognitive process. But in the programming context, and in the area of the software pattern concept itself, clearly the analysis and synthesis processes are central. The aim of the analysis of a problem situation is to discover the patterns revealed in the distribution of the forces and constraints within the situation. Of course this needs to be done on the basis of a knowledge and comprehension of the programming resources available in the domain.

The creative part of programming involves matching the results of the analysis of the problem with the appropriate resources to build the new synthesis of the forces that solves the problem. Because this is a pattern matching process that involves identifying the patterns in the situation with the patterns of syntax available in the programming environment, the idea of presenting the resources in a specialised format to assist the matching process arises. And because of the association with pattern matching, this specialised format has come to be known as a pattern, in the specialised sense of design or software pattern.

## 2.3 Patterns in Teaching Programming

Another way of looking at the results of applying Bloom's taxonomy to the programming domain is that it points to the practical aspect of the field. The design patterns show up in the categories that are most clearly involved in the programming activity itself. In learning this activity then, patterns will be mainly useful in that they can be seen as a way of encapsulating knowledge in a form useful to the activity.

This is, probably, the most powerful aspect of design patterns in the purely pedagogical sense. It is doubtful that patterns, or at least those based on programming language features, have much pedagogical power in themselves. They are just, more or less, the same basic material presented in a systematic, standardised format. But for the practical aspect of solving programming problems, one would probably not bother to write programming language constructs into the design pattern form.

So we don't see the pattern language's main use as a teaching tool or methodology. Our view is that you simply present the patterns in the teaching material in the places where examples are normally used. One of the 'discoveries' made in this project, is that all the elements of a pattern are there in the examples used in course material already. They are just not presented in a systematic way.

## 2.4 Solving Problems

The idea of systematising the examples used in course material into pattern format is to provide the extra benefits of the pattern form in the problem solving situation, Bloom's fourth category, not primarily to help students understand the material (the earlier categories). The skill of solving problems grows out of practice, and patterns are directly useful at this level. Examples are important here, but novices mostly fail to pick up their 'essence'. They attempt to apply them in a rigid manner, failing to grasp the general principle that lies at the heart of the example.

Patterns would appear to express the general nature of the problem-solution nexus much more clearly. Moreover, they provide 'slipperiness', that is they make the example more easily exportable from one context to another, so that it can move across domains like analogies do in natural language.

Design patterns are, to all intents and purposes, analogies in the programming domain. In a sense they are attempting to highlight the general nature of the example - in effect they help to hide the code. Seeing 'the code' as 'the example' is a large part of the difficulty in applying it in similar, but different, situations. The pattern form sets 'the code' in a more realistic context, thus generalizing it. But it is the contextual relationships between the patterns, the pattern language, that provides the main advantage of patterns, the generative power of process.

## 2.5 Pedagogical Implications

Bloom's taxonomy suggests that learning takes place in stages. Comprehension requires knowledge, application requires comprehension, and so on. Learning to program is particularly difficult as it requires high-level cognitive activities such as analysis and synthesis early. Given this, the cognitive level of a particular activity cannot be the determinant of its place in the learning process.

Students of programming need to be given an appreciation of synthesis, the putting together of base concepts, right from the start. The solution in programming seems to have been, up to now, to give the students experience with small parts of programs rather than expecting them to write whole programs from scratch (Buck & Stucki 2000). The material that needs to be understood for the application and analysis levels must be introduced in small increments to allow the student to concentrate on acquiring the real skill of programming - solving problems.

## 3 The Pattern Process

### 3.1 Language and Context

The pattern approach requires more than just codifying solutions to recurring problems. Capturing the essence of recurring problems in a standard form is the base idea, but programming problems are too large to be solved using a single pattern.

What the pattern language provides is a view of a system that derives from prior experience in using it. It shows how patterns come into play while avoiding the need to closely analyse the forces pertaining in the context of the problem.

As a pattern is applied it changes the dynamics of the situation. The problem, its context, and the forces in it, are transformed by the pattern because, not only has the problem itself been modified, but the context of the problem has been altered by the interaction with the context of the pattern in the pattern language. Applying one pattern has brought other patterns into play.

In other words the pattern has brought its context in the language with it and thereby transformed the context of the problem. This is because the place of the pattern in the language is a reflection of prior experience in solving the particular problem that it encapsulates. In dealing with situations in the domain, others have discovered the particular structure of relationships between the patterns.

So the pattern language structure is encoding the underlying order in a powerful way. It is, in a sense, a map of the common problems in the system. Any problem currently being dealt with will have its context in the system revealed by the patterns to which it is related.

The language not only connects the patterns to each other, but helps them to come to life, by giving each one a realistic context, and encouraging imagination to give life to the combinations which the connected patterns generate.

(Alexander 1979, p.315)

### 3.2 Patterns in Pedagogy

In our case the system involved in the exercise is that of the programming language being used in the teaching process. Hence the basic syntactic elements of the programming language provide many of the patterns because any problem to be solved must be solved using these constructs. But solving a problem requires

thinking at a higher level than is dealt with by the syntactic elements alone; it involves designing a solution as well as implementing it.

The point about the process that derives from using a pattern language is that it results in a design, a sequence of patterns, not a solution in program code. This means that in writing a pattern language you have to be concerned about design decisions that are above the level of syntax; concepts, like repetition and choice, that give the pattern language more structure than a programming language alone has.

Patterns are, in an important sense, an attempt to bridge the qualitative experience gap. As a pattern writer you are trying to give the pattern user a sense of why you chose this particular solution. Solving a problem involves a cognitive process that has little or nothing to do with the arbitrary structure of the programming language.

The spirit of the pattern idea is an attempt to capture the experience of solving a problem in a more general, less rigid, form. The design represented by a sequence of patterns is independent, to some extent, of a particular syntax. The idea of basing a pattern language on the syntax is therefore a compromise based on the lack of syntactic experience of the novice. Teaching people to program involves both the cognitive problem solving process and the syntactic structure of the programming language. There is, inevitably, a degree of conflict here.

Nevertheless the basic force of the idea remains. The patterns encode the experience of solving a particular type of problem. The pattern language reveals the context of each pattern and by this means provides the process by which solutions to larger problems are built. Context is everything, especially to the novice, because it points to what happens next.

## 4 Testing the Pattern Process

### 4.1 The Proposal

To test the process that arises from using a pattern language we propose to give two groups of students access to different teaching materials. Both groups will then attempt a common programming assignment under the same conditions so that the effectiveness of the two sets of materials can be compared. One group will be introduced to a set of patterns and an associated pattern language, while the other group will be given material that covers the same ground but is written in a non-pattern form.

Since the patterns group would need both the patterns themselves, and the pattern language diagram that illustrates the contextual relationships between them, we also propose to test the use of the pattern language idea itself. In surveying the situation in Computer Science it is clear that, by and large, patterns have been adopted in isolation from pattern languages. We can find very few actual pattern languages in the field. This means that the main benefit that Alexander saw in his pattern work, the generative power of the language is missing.

As in the case of natural languages, the pattern language is generative. It not only tells us the rules of arrangement, but shows us how to construct arrangements - as many as we want - which satisfy the rules.

(Alexander 1979, p.186)

By arranging a third group, who would be given the same patterns as the first group, but not the pattern language, we could effectively test this notion of the generative power of the pattern language. This

widens the usefulness of the experiment beyond the educational domain into a study of the use of patterns in software development generally.

As Alexander himself has said, while addressing a gathering of several thousand software development people at the OOPSLA conference in 1996:

I think that insofar as patterns have become useful tools in the design of software, it helps the task of programming in that way. It is a nice neat format and that is fine. However, that is not all that pattern languages are supposed to do. The pattern language we began creating in the 1970's had other essential features. First, it has a moral component. Second, it has the aim of creating coherence, morphological coherence in the things that are made with it. And third, it is generative: it allows people to create coherence, morally sound objects, and encourages and enables this process because of its emphasis on the created whole.

(Alexander 1999)

If Alexander is correct, the widening of our experiment should demonstrate this in practice. The group with both the patterns and the pattern language should perform better in a programming test than the patterns only group because of this generative power of the language.

## 4.2 Practical Considerations

The implication of subjecting people to a programming test is that they have some basic level of proficiency in a programming language. This means that such a test would need to be given at the end of the first programming course. In our situation, the assessment of the first programming topic is largely, if not entirely, a measure of the students' understanding of Java. We can therefore relate performance in a programming test to the level of proficiency in the programming language demonstrated in the first course results.

Given this, the obvious strategy is to test students near the beginning of the second programming course. This allows us to assume a basic level of familiarity with the language, and to use the first course results to control for various degrees of familiarity. We want to test programming skill, not knowledge.

Another consideration is that while the material is not entirely new, there is nevertheless a significant amount of learning required to assimilate the new format and enable its use in solving programming problems. So as well as the pattern material there would need to be a process of introducing it. We decided to provide an introductory lecture and a follow up discussion based on stepping through a couple of examples. The second session would be driven by student participation.

In order to keep the group experiences similar, introductory sessions for the other two groups would also be needed. Doing this for the group that has no pattern exposure is not too difficult. However the group with the patterns but no language would have to be carefully dealt with, in order not to introduce the pattern language inadvertently in explaining the use of the patterns.

## 4.3 The Need for a Trial

Since skill acquisition is a complex and poorly understood process, an experiment to test a skill is quite likely to overlook some factors. Testing human subjects is notoriously difficult. As non-experts in the

field of Social Science, we thought it best to run a pilot study to uncover any shortcomings in the design of the experiment, and to test the basic feasibility of the idea. A full account of the trial, including the materials used in the experiment is available elsewhere (see url in Conclusion). A brief description of the trial and its results follows.

## 5 A Trial Run

### 5.1 Experimental Design and Setup

As discussed above, we decided to run the experiment in a fairly informal manner to test its feasibility. To this end, a pool of volunteers from the cohort of students currently undertaking the second programming topic was recruited.

There was a short session during the third lecture in the second programming topic to introduce the project and take expressions of interest. No inducement other than a possible advantage in acquiring the skill of programming was offered. Because of this we expected that the idea would appeal mainly to those who were aware that they were struggling with programming. Of the approximately 300 students, 84 signed up as being interested in taking part in the experiment. As expected, this number fell off as further steps in the process were taken, so that by the time the trial began 27 remained.

The pool of volunteers was broken down into two smaller groups, a test group of 13, Group A, and a control group of 14, Group B. The test group was given a set of patterns based on the syntax of Java and a pattern language structure in the form of a diagram. The control group was provided with material describing the same basic programming language details that are covered in the patterns, but structured in non-pattern form. This material was loosely based on the course material provided to first programming course students from 1998 to 2001.

The groups were arranged on the basis of the results of the participants in the first programming course, so that each group contained a similar spread of ability. Some people did not turn up for their introductory sessions, so the participation rate was further reduced to 18.

The introductory lectures were based around a discussion of the material and an analysis of the process of using it to solve a simple programming problem. Another simple programming problem formed the basis for the workshop sessions and these were driven by student participation.

A week later both groups completed a programming assignment. The programming assignment was taken on an individual basis under the conditions that normally apply for on-line exams in our school. The program code produced was later assessed for the degree of completion reached and on the basis of its quality in terms of programming technique and style.

The programming task set was divided into six stages of progressive difficulty, designed so that anybody who had passed the first programming course should have little trouble achieving stage 1, and that the increasing difficulty of the subsequent stages would cause a spread of achievement that could be used to measure the effect of the different materials.

For the control group, we anticipated the correlation between the first course result and the stage reached to be something like that shown in Table 2. Any beneficial effect based on the effectiveness of different materials should show up by shifting the achieved ranges upwards.

Topic1 result	Expected spread of achievement
HD	stage 5 or 6
DN	stage 4 or 5
CR	stage 3 or 4
P	stage 1 or 2

Table 2: Expected range for the control group

## 5.2 Results

Table 3 summarises the results achieved in the test by the individual participants. The number in the result column is the stage reached within three hours from the start time. Coding style quality is assessed in column 5. The participant ID was assigned randomly in order to disguise the real identity of the students, and the starting letter identifies the group to which the participant was assigned - A for the pattern group, and B for the control group.

ID	Topic1 result	Stage reached	Style
A1	P	2	satisfactory
A2	CR	5	very good
A3	HD	6(almost)	excellent
A4	CR	4	good
A5	P	1	satisfactory
A6	DN	5(with a bug)	good
B1	HD	5	very good
B2	CR	4	good
B3	CR	1	good
B4	P	2	good

Table 3: Summary of individual results

Figure 1 plots the results achieved in the test, using circles for members of the control group and crosses for pattern users, against the range expected from CP1 performance. The expected range is indicated by lines.

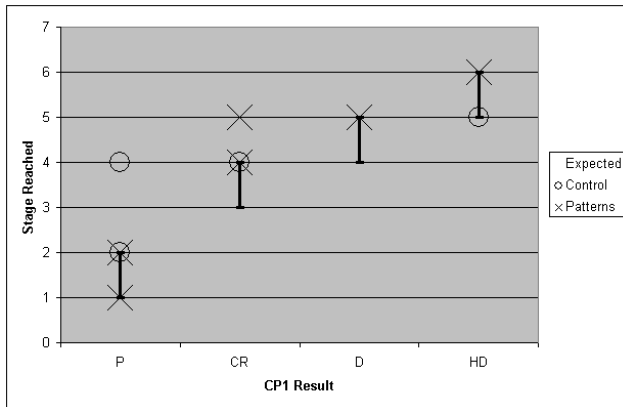


Figure 1: Comparison of results with expected range.

## 5.3 Discussion of the Results

Figure 1 shows that the individual results for the control group matched the expected performance except that one student achieved stage 4, better than the expected stage 1 or 2. The results for the pattern group tended to be in the high end of the range expected, except for one student who achieved a stage 1 result. Some variation in this first programming course result level is not unexpected because the Pass grade covers a larger range of marks than the other grades.

The number of students who attempted the programming task is too small to allow any statistically

significant interpretation of differences in achievement levels between the two groups. However the exercise was highly useful in providing indications of how the study should proceed. For example, it was observed that the people in the patterns group did not, by and large, use the pattern language or the patterns. (The control group, also, did not appear to use the material provided). If the materials were really affecting performance we would expect the use of the materials by the participants to be obvious during the exercise. It is possible that the people exposed to the pattern material had internalised it by the time of the test, but this is considered unlikely.

The apparent failure to use the materials provided indicates that testing the usefulness of patterns in this way will require a substantially different approach than was tried here. In particular there would need to be some way of 'forcing' the students to complete the design stage before programming. For example, students could be required to hand up some sort of documentation other than the program code. The implications of this for the design of the experiment are that the level of effort, and hence time, required of participants will be higher. Trying to build any sort of design process into the programming practice of novice programmers will require more than a simple introductory session or two.

## 5.4 Lessons

We feel that the results of the trial indicate that it is possible to measure the effects of different materials on performance in this way, but that any subsequent experiment will need to find a way to increase participation levels. Although 84 people from the 2002 cohort (approximately 300) expressed an interest in taking part in the experiment, difficulties in arranging suitable times quickly reduced this to 27. Eighteen people attended the introductory sessions, but only half of these made it to the programming test.

And somewhat in contradiction to the need to increase the active participation rate is the fact that the experiment indicates that the approach of merely introducing people to the materials and expecting them to be used in a programming test is not realistic. That is, there is both a need to increase the length of the time that volunteers commit to the study, and a need to find a way of increasing the number of volunteers completing.

## 6 Conclusion

The trial run demonstrated the basic feasibility of measuring the effectiveness of different materials. The correlation between the understanding of the programming environment, indicated in our case by the first course results, and individual performance in the programming test did hold as expected. Therefore given the correct experimental setup, the experiment would clearly indicate any difference in the effectiveness of the materials being used.

However, it showed that there are two factors that would need to be addressed in any full scale study: the drop in participation rates over time and the failure to use the materials supplied. The obvious solution to the first issue is to offer payment for involvement. However this raises the difficulty of people volunteering just for the money and not putting any real effort into the experiment. In order to produce meaningful results a level of commitment to the project is required.

The participation rate issue can be addressed, perhaps, by the following procedure. At the introductory session, we ask for people to express an interest only.

No mention is made of any payment at this stage. We then offer a payment to those people who have expressed an interest at a follow up meeting. This should avoid the problem of people volunteering just for the payment.

However the question about real commitment remains. On its own, this mechanism does not appear to say anything about levels of commitment. Unfortunately we don't really know what factors were involved in causing the attrition of volunteers. Offering payment at the follow up meeting assumes that the main factor was time rather than lack of commitment. That is, people had to make a judgement about spending time on a project that offered them nothing but a vague possibility of some extra insight into programming as against all the other pressing needs of life. So the real problem at this level, commitment, is probably not addressed.

This concern feeds into the second difficulty that arose during the first experiment, the fact that people did not seem to be using the material provided. We could perhaps force the use of the material issue, to some extent, by requiring an initial design document. However the difficulty again is that we don't really know why the students didn't use the material. In forcing a design stage we are assuming that they didn't use the design material because they are so used to programming without doing design that it just doesn't seem relevant.

Forcing a design stage would therefore seem to be negative in terms of commitment value. We can force students to produce a design document, but does that really ensure that they use the material in a way that will give us a meaningful measure of the usefulness of the material? We need to think about this commitment issue very carefully. If the start-with-coding syndrome really is a habit that is picked up by students as a result of the way that the initial exposure to programming occurs, then it is difficult to believe that we can overcome this just by forcing a design stage. Maybe we have to think about ways of trying to prevent the habit developing in the first place, as well as ways of preventing it impacting on the experiment.

In testing changes in teaching practice it is always important to gather results from a range of different environments. And other experimenters will doubtless come up with ways of proceeding that we have overlooked. The materials developed in this project will be made freely available to anyone who wants to attempt an experiment like this. They are available at this site: [www.infoeng.flinders.edu.au/papers/20030011.pdf](http://www.infoeng.flinders.edu.au/papers/20030011.pdf)

Probably the most significant finding of the trial was that concerning the structure of the first programming course. If we believe that designing a solution before coding is desirable then this needs to be made a primary aim of the first programming course. The trial run demonstrated that introducing the concept of design-before-code is not something that can be done after the learning of the programming language. We need to be clearer about what the aims of the first course are, so that we can be sure that they are addressed in the course material.

## 7 Acknowledgements

Joe Bergin of Pace University suggested the idea of attempting to measure the effectiveness of patterns in the novice programming environment. The co-operation of the topic co-ordinator for CP2A, Graham Roberts, was vital in the running of the trial experiment. We also wish to acknowledge the time and effort put into this project by the volunteers.

## References

- Alexander, C. (1979), *The Timeless Way of Building*, first edn, Oxford University Press, New York.
- Alexander, C. (1999), 'The origins of pattern theory', *IEEE Software* **16**(5), 71 – 82.
- Bergin, J. (n.d.), 'Joseph bergin home page', <http://csis.pace.edu/~bergin/>.
- Bloom, B. S. (1971), *Taxonomy of educational objectives : the classification of educational goals : handbook 1 : Cognitive domain*, David McKay Co., New York.
- Buck, D. & Stucki, D. J. (2000), Design early considered harmful: graduated exposure to complexity and structure based on levels of cognitive development, in 'Proceedings of the thirty-first SIGCSE technical symposium on Computer science education', ACM Press, pp. 75–79.
- Clancy, M. J. & Linn, M. C. (1999), Patterns and pedagogy, in 'The proceedings of the thirtieth SIGCSE technical symposium on Computer science education', ACM Press, pp. 37–42.
- Daniels, M. Petre, M. & Berglund, A. (1998), 'Building a rigorous research agenda into changes in teaching', At <http://www.docs.uu.se/docs/cse/papers/brisbane98.html>.
- East, J. P. & Wallingford, E. (1997), Pattern-based programming in initial instruction (seminar), in 'Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education', ACM Press, p. 393.
- Haverty, L., Koedinger, K., Klahr, D. & Alibali, M. (n.d.), 'Solving inductive reasoning problems in mathematics', [www.cognitivesciencesociety.org/abstract/haverty.html](http://www.cognitivesciencesociety.org/abstract/haverty.html).
- Lister, R. & Leaney, J. (2003), First year programming: Let all the flowers bloom, in T. Greening & R. Lister, eds, 'Fifth Australasian Computing Education Conference (ACE2003)', Vol. 20 of *Conferences in Research and Practice in Information Technology*, ACS, Adelaide, Australia, pp. 221–230.
- Mayer, R. E., Dyck, J. L. & Vilberg, W. (1986), 'Learning to program and learning to think: what's the connection?', *Communications of the ACM* **29**(7), 605–610.
- Porter, R. & Calder, P. (2003), A pattern-based problem-solving process for novice programmers, in T. Greening & R. Lister, eds, 'Fifth Australasian Computing Education Conference (ACE2003)', Vol. 20 of *Conferences in Research and Practice in Information Technology*, ACS, Adelaide, Australia, pp. 231–238.
- Reed, D. (1998), Incorporating problem-solving patterns in cs1, in 'Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education', ACM Press, pp. 6–9.
- VanLengen, C. & Maddux, C. (1990), 'Does instruction in computer programming improve problem solving ability', *Journal of IS Education* **2**(2), 47–49.
- Wallingford, E. (2001), 'The elementary patterns home page', At <http://www.cs.uni.edu/~wallingf/patterns/elementary/>.