

A Pattern-Based Problem-Solving Process for Novice Programmers

Ron Porter and Paul Calder

¹ 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. This paper suggests a process for applying patterns to problems that derives from the relationships between patterns in a pattern language, and demonstrates its use in a simple problem. This approach suits the pedagogical model based on Bloom's taxonomy as it enables the required knowledge to be partitioned and dealt with in stages.

Keywords: patterns, learning to program, problem solving.

1 Introduction

The pedagogy based on Bloom's taxonomy of cognitive development suggests learning in stages, each stage building on the knowledge gained in the previous stage. Comprehension requires knowledge, application requires comprehension, and so on. Unfortunately, programming presents a problem in this respect as it requires high-level cognitive activities such as analysis and synthesis early in the learning process. The staging therefore needs to be conducted on a partitioning of the material based on something other than cognitive levels. The solution in programming is 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. The pattern approach is a way of encapsulating knowledge about problems and their solutions that is simple, concise, and direct. Moreover the organisation of the knowledge in the pattern language form adds benefits in terms of synthesis and evaluation as will be discussed here. Patterns thus have two benefits at the level of novice programming: They suit the pedagogy based on Bloom's taxonomy, and they address the real issue in programming - the solving of problems.

The power of the pattern concept is based on the solving of problems using known solutions. Problems tend to recur in many different situations, and it is

the repetition that makes them patterns that allows them to be used as a basis for design. The software pattern concept is based on this simple fact of recurrence. Since the problem-solution combination keeps recurring, the idea is to design an artifact, the pattern, that captures its essence in a standard form. However, since most programming problems are too large to be solved using a single pattern, the pattern concept must include the facility for patterns to be put together to form a 'pattern sequence', that is, a particular configuration of patterns that solves a particular problem.

Fortunately, the patterns in a particular domain form a powerful network of connections in terms of the context in which they appear. For example, if you are building or designing a wall you need to think about other things, like doors, and maybe even windows. Thus `wall` forms the context in which you can expect to find `door` and `window`. This network of relationships between the patterns in a domain gives the overall collection a degree of coherence and forms the structure that is known as a pattern language. The use of the word 'language' is suggestive here because of the contextual meaning implied by the relationships between the patterns. Solutions for larger problems can be constructed from the individual patterns because of the connections of meaning between them, just as the relationships between words allow the construction of larger concepts through combination.

A pattern language, then, represents the design space as a network of the patterns, and therefore the problems, that occur within it. As architect Christopher Alexander says;

the real work of any process of design lies in this task of **making up the language** (emphasis added) from which you can later generate the one particular design. You must make the language first, because it is the structure and the content of the language which determine the design.

(Alexander 1979, p.324)

The language provides the dynamics for generating designs just as a natural language provides the dynamics for generating written artifacts. It provides a means for developing sequences of patterns.

To get it [a sequence], a static pattern language ... was then re-stated as a generative sequence. In its sequence form it shows the user the process of unfolding, in sequence, in such a way as to allow a good building to be made, very easily, for the particular conditions of a given site.

(Alexander 2002, p.303)

In approaching a programming problem, the idea is to analyse the context of the problem with a view to understanding the forces or constraints that will shape the solution. Once this is done, the forces so exposed provide the means to begin construction of the 'pattern sequence' for the solution. The advantages of the pattern approach are that the configuration of forces in a given problem situation can suggest which pattern best matches, thus providing an element of synthesis within the analytic process, and that the 'pattern sequence' for the solution can be constructed without having to deal with the code directly. The pattern language drives the sequence-producing process, which specifies the coded solution.

This is also the advantage of patterns over other methods, such as pseudocode, flowcharts, Nassi-Schneiderman charts and the like. In these cases the solution is generated by the programmer rather than being, partly at least, driven by the relationships between the concepts that are illuminated by the pattern language. The pattern language is a more powerful abstraction. It doesn't just provide a set of symbols to be manipulated by the designer; it adds a view of the context, the relationships between the concepts, to the design space.

Being a problem-solution pair, patterns have attracted notice as being potentially useful in the context of learning to program, and many people have explored this potential. In particular, Joe Bergin (Bergin 2002) has become the focus for a lot of work in the field of patterns in pedagogy. Eugene Wallingford (Wallingford 2001) is another person in this field, and many others have contributed as well. Most of this work is in the form of patterns, but there are works that the authors describe as pattern languages in progress, and some discussion about the idea of using patterns in the teaching process.

The motivation for this paper is to explore the process of using patterns in a pattern language to specify the benefits that may be expected in the novice programming environment. To this end we have written a set of patterns based on low level programming language features and this paper explores their use in solving a simple problem. Our aim is to demonstrate that the advantages in the pattern concept for high level design are also pertinent to the solving of problems at the novice level of programming.

Before trying any idea like this in practice there needs to be some basis in theory for expecting it to work. The aim of this paper is basically that first step, an attempt to uncover possible mechanisms within the pattern language concept on which a problem solving process for novices could be built. In its original setting, building architecture, the pattern concept always had a strong sense of process built into it, but this seems to have been lacking in the software pattern field. (Alexander 1999) This paper suggests that process does flow from the use of pattern languages in software development and attempts to demonstrate, at a theoretical level, the benefits that can be expected to accrue to novices. The patterns and the process of using them are directed at providing novice programmers with a viable way of tackling new problems, the area in which most struggle.

2 Applying Patterns to a Problem

Novices need to be given a clearly defined notion of process because most of the difficulties at this level arise from not knowing how to apply knowledge, rather than from a lack of knowledge. The pattern form associates a problem and a solution, and provides information about the context in which the pattern is applicable and the forces that it resolves.

This suggests that analysing the problem situation in these terms should provide a way of thinking about the problem in terms of patterns, which should help point towards a solution. The example section illustrates the use of the process.

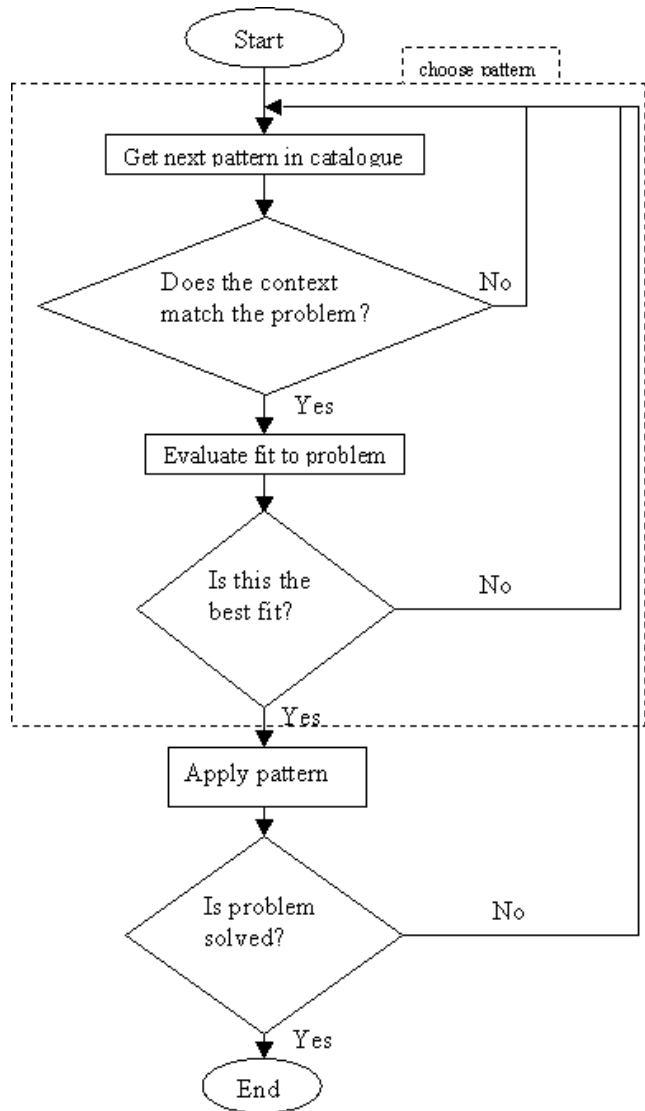


Figure 1: Building the 'pattern sequence'

Figure 1 illustrates the development process. The steps contained in the dashed rectangle attempt to identify the pattern in the catalogue of patterns that 'best fits' the context and forces of the problem. Applying this pattern modifies or refines the initial problem as shown in Figure 3.

One of the difficulties for experts in instructing novices is that the expert will make unconscious assumptions about a process like this. The expert recognizes the patterns in a situation almost without thinking, and therefore doesn't explain the entire process when teaching it. As Sleeman pointed out (Sleeman 1986), most people have great difficulty in explaining the process of finding the largest of a set of integers because they do it without consciously thinking about how it actually happens. In this case the expert, and even novices as they progress, will tend not to have to explicitly iterate through an entire catalogue of patterns in order to match a pattern to a problem. Instead the pattern selection processes will tend to merge into a single step, "choose the pattern that best advances the solution of this problem". Merging the dashed box step, "choose pattern", with the next step, "apply pattern", creates an overall pro-

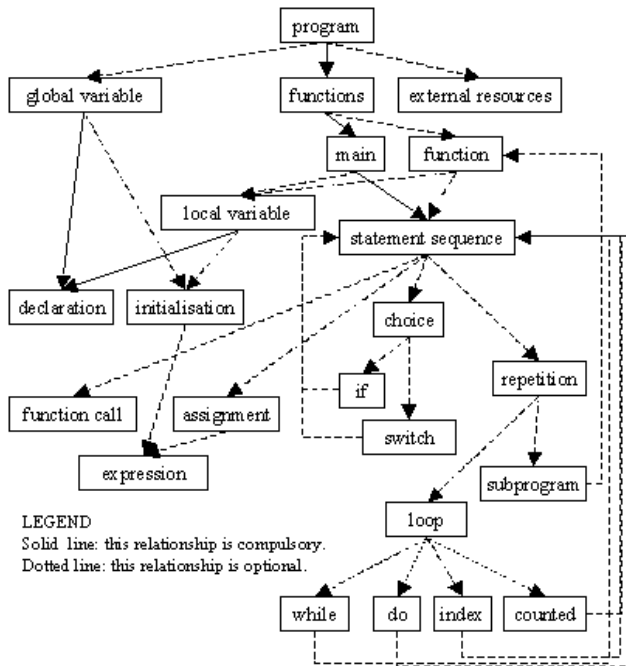


Figure 2: The Pattern Language

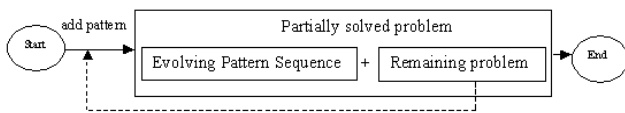


Figure 3: Adding a pattern to the sequence

cess to be called 'add pattern' in Figure 3.

There are three elements to the problem solving process. The first is the pattern language diagram (see Figure 2), which provides the context for the next pattern to be applied. An indication of the forces resolved by applying it is garnered from the 'remaining problem' section of the 'add pattern' process illustrated in Figure 3, which is the second element. Further details about applying the pattern are available, if necessary, in the pattern itself. The third element, the context for the next pattern, is derived by following the arrows in the pattern language diagram downwards from the previously applied pattern. A single solid arrow means that the pattern to which it points will always be applicable, while arrows with dashed lines indicate that the context involves making a choice between several patterns. The semantics of any downward pointing arrow is that following it to its target 'adds structure' to the pattern at its source. A pattern will remain open until there is no more 'structure' to be added to it, at which point it becomes complete. Note that some arrows lead back to patterns higher in the diagram, indicating recursion in the language. For example, a loop will have a block of code that is to be repeated. This block needs to be built up from statement sequence like any other, and the upwards arrows cover this situation.

The process of solving a problem, therefore, takes the following form:

1. Ascertain the forces in the unresolved part of the problem from the 'remaining problem' section of the 'add pattern' process.
2. Find the context of the next pattern in the pattern language diagram.
3. Apply the pattern with the help of the details in the pattern itself.

This interplay between the three elements drives the process that results in the building of a sequence of

patterns that describes the solution.

Identifying the context of a problem is a matter of knowing the location of the last applied open pattern (that is, the pattern that still needs structure added to it). This is an indication of where you are in the development of your solution. The use of context, therefore, involves following the arrows from previously applied patterns. That is, the context for a pattern to be applied is a pattern that is still incomplete, that still requires additional structure added. In this case it is statement sequence again, unless it is clear that the current sequence is complete. If it isn't, the context is the same as before, the four patterns below statement sequence, and we then refer back to the current state of the problem solution to study the remaining problem for guidance in making a choice between them. This shows that in applying the 'add pattern' process in Figure 3, the remaining problem is the source of an indication of the forces in the situation, and the position in the pattern language provides the context for the next pattern. The movement back and forth between the two diagrams is the relating of forces and context, and this is what drives the problem solving process, building the 'pattern sequence'.

The subsequent steps repeat the process of selecting and applying further patterns until the solution is completely specified and no more refinement is necessary. The pattern sequence for the solution has thereby been constructed. Note that the process does not specify when coding is done. It can be done at any stage if the solution up to that point needs testing. Otherwise it can be left until the pattern sequence is complete.

3 A Simple Example

This section works through the process of solving a simple problem using patterns. It attempts to enumerate those things that are always done when solving a problem. That is, we are looking for the activities that are common to the problem solving process. So in that sense we are building a pattern view of the problem solving process.

The Problem Specification.

Write a program to produce a multiplication table for integers from 1 to 9 as shown below.

```

1
2 4
3 6 9
4 8 12 16
5 10 15 20 25
6 12 18 24 30 36
7 14 21 28 35 42 49
8 16 24 32 40 48 56 64
9 18 27 36 45 54 63 72 81

```

Producing the above output on the computer screen is the initial problem. Applying the process illustrated by Figures 1 and 3 to this initial problem will divide the problem into two parts, the solved and unsolved part. The unsolved part, 'remaining problem' in Figure 3, is then put into the 'add pattern' step to again divide the problem. Continued iterations of this process will generate a sequence of patterns that 'describe' the solution. The iterations continue until such time as there is no more 'remaining problem'. The remainder of this section is a description of this process. Some iterations are insignificant in terms of understanding how the process works, and are subsumed as elements of one larger iteration. In these cases the new 'pattern sequence' diagram pro-

duced will contain several new patterns instead of the normal one.

3.1 First iteration of 'add pattern' process

The problem specification states that the task is to write a program to produce some output on the screen. Thus the first pattern to be indicated is `program`. This pattern indicates that it is the appropriate choice when the problem is suitable to a computer based solution and it specifies the structure of a program in terms of C syntax.

Pattern 1 : `program`

3.2 Second iteration of 'add pattern' process

Referring to the location of `program` in the pattern language diagram gives us the context for the next pattern, which involves a choice between three possibilities, one of which, `functions`, is compulsory. In a situation like this, where there is a combination of optional and compulsory lines of development, the optional ones should be investigated first, and referring back to the problem specification suggests that some external resources in the form of the output routines in the input/output library are all that is indicated by these. Taking the compulsory arrow to `functions` involves making a decision about the nature of the program. Thus, so far, the solution involves a sequence of three patterns.

Pattern 1 : `program`
Pattern 2 : `external resources`
Pattern 3 : `functions`

3.3 Third iteration of 'add pattern' process

The patterns in the context formed by `functions` are `function` and `main`. In reading the 'remaining problem', no need for a separate function suggests itself at this stage. This leaves `main` where the actual execution of the program will begin, thus involving the next pattern `statement sequence`. The patterns applied so far are the mainly mechanical actions that set up the programming environment, and applying them has not materially advanced the development of the solution. `statement sequence`, however, sets the context for the attacking of the problem, and offers four ways of doing this. Making the choice between the four patterns below `statement sequence` means making an analysis of the forces in the remaining problem for clues.

Pattern 1 : `program`
Pattern 2 : `external resources`
Pattern 3 : `functions`
Pattern 4 : `main`
Pattern 5 : `statement sequence`

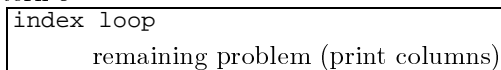
3.4 Fourth iteration of 'add pattern' process

The repetitive nature of the output required is the force acting in this situation, which suggests that the program control should consist of repeated action. Fundamentally the task is to generate a specific set of output lines, corresponding to the rows of the table. This problem is addressed by the `repetition` pattern, which tells how to perform a series of actions, in this case "print a line", several times in succession. `Repetition` offers a choice between `loop` or

`subprogram` and the pattern indicates that, as the repetitions are, in this case, contiguous, `loop` is the choice we should make. The `loop` pattern shows how the division of the program into a repeating and a non-repeating part is represented in a sequence, and gives several variants. In this case, the best match is with the `index-loop` form, which is appropriate when the action to be performed varies for each repetition, and the variation can be expressed by a series of values. Specifically, the loop will execute for each of the values, 1 to 9, each line displaying multiples of those values.

For this problem, the entire task is contained in the code to "print a line"; there is nothing left of the original problem that is not repeated. Reference to the `loop` pattern tells us that therefore the rest of the sequence will be contained inside the loop as there is no non-repeating part to go after the loop. The box in the `loop` pattern contains the repeating block - if the remaining problem suggested any need for further non-repeating code, the pattern sequence for this would continue after the box.

Pattern 1 : `program`.
Pattern 2 : `external resources`.
Pattern 3 : `functions`.
Pattern 4 : `main`.
Pattern 5 : `statement sequence`.
Pattern 6 : `repetition`.
Pattern 7 : `loop`.
Pattern 8 :

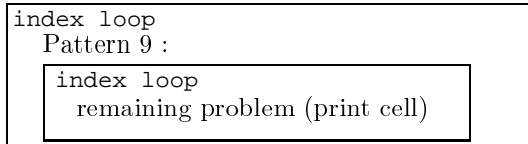


3.5 Fifth iteration of 'add pattern' process

The repeating part of the problem will involve setting up a separate sequence of java statements contained in the block of code that is to be executed multiple times. This fact is indicated by the arrow that leads upwards to `statement sequence`, from where the repeating block can be developed. This provides the context for the first pattern in the repeating block. The remaining problem shows the need to examine what is required for each print line statement in the loop. The line is made up, conceptually, of a series of columns. Just as the `loop` pattern provided the answer to repeating rows it is clear that repeating columns probably requires a similar approach. That is, each line will be a series of columns printed out by an `index loop`. Proceeding through `repetition` and `loop` to `index loop` is the same sequence as before, so here it is telescoped into the one pattern, `index loop`.

This gives us the overall structure of the solution, which now takes the form shown below.

Pattern 1 : `program`.
Pattern 2 : `external resources`.
Pattern 3 : `functions`.
Pattern 4 : `main`.
Pattern 5 : `statement sequence`.
Pattern 6 : `repetition`.
Pattern 7 : `loop`.
Pattern 8 :



3.6 Sixth iteration of 'add pattern' process

Hence we have reduced the problem to "print cell", which in turn reduces to the problem of identifying and producing the number that makes up each cell. In a multiplication table, each cell contains the product of the row number and the column number. That is, each cell contains the formula `row * column`. In this case the loop control variables directly represent these values, so the "print column" sub-problem becomes:

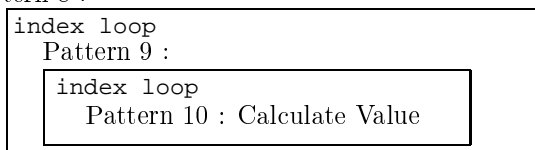
```
print (outer index * inner index)
```

One obvious feature of the required output that has not been considered yet, is the fact that each successive line increases in length by one column. The number of columns in each line is constrained by the line number.

```
for (j = 1; j <= i; j++)  
where j controls the inner loop, and i the outer loop.
```

In a situation like this, if the developer is confident that the rest of the task is trivial, the development of the pattern sequence can be stopped at this point and a placeholder used to indicate that there is more to be done to complete the sequence. In this situation we will use a placeholder called 'calculate values' as Pattern 10. However, if necessary, the process can be continued through declaration, assignment and expression patterns to find out how a dependent value can be produced by an expression. Calculating the values completes the solution of the initial problem because applying it leaves no smaller problem to be solved. If the nine patterns are then put together in Alexander's "pattern sequence" form, the following pattern sequence results.

Pattern 1 : program.
Pattern 2 : external resources.
Pattern 3 : functions.
Pattern 4 : main.
Pattern 5 : statement sequence.
Pattern 6 : repetition.
Pattern 7 : loop.
Pattern 8 :



This shows the structure of the solution as Alexander says it would - "in its sequence form it shows the user the process of unfolding, in sequence, in such a way as to allow a good building to be made, very easily, for the particular conditions of a given site." In our case it is not a 'building', of course, but the 'nested loop' structure of our solution, with the inner loop containing the number sequence generating pattern, is clearly evident in the diagram. Finally, the completion of the solution involves providing the code to fit the design, and the 'code example' sections in each pattern provide a guide to this.

4 Discussion

The patterns used at this level are, more or less, the examples that would normally be used in a first programming course, packaged in the pattern format. The advantages of this approach are twofold. Firstly the pattern format generalises the example code by

providing extra information about its use in different contexts. Secondly, it separates the pattern from the other teaching material that surrounds it. This separation can be optimised by reproducing the patterns in a separate document for ease of use in the assignment situation. In this way the pattern format addresses the two main problems that novices have in using the examples provided in their course material, finding them and using them. The patterns are written during the normal course of developing the other materials for the first programming course. They are software or design patterns in the sense of the patterns in the book "Design Patterns: Elements of Reusable Object-Oriented Software" (Gamma, Helm, Johnson & Vlissides 1995), and are only pedagogical in the sense that they are used in a teaching environment.

The advantage of adding patterns to the design process is threefold. First, it enables the tackling of coarser grained features in the original problem specification than would be possible otherwise (that is, there will be fewer steps in the decomposition process). Having identified a pattern that handles part of the problem you can effectively ignore that part of the problem and concentrate on the smaller problems exposed by the first pattern.

The second advantage is that the patterns discovered by this process will produce the 'pattern sequence' effect, which facilitates the design of the solution as shown by the evolving sequence of patterns through the successive iterations. The relationships between the patterns in a pattern language provide the connective power that enables sequences of patterns to solve problems larger than those solved by individual patterns in a pattern language. This is a case where the whole is more than the sum of its parts. The 'pattern language' enables large problems to be dealt with by building up a sequence of patterns.

What the network of relationships, illustrated in the pattern language, adds to the process, is an indication of the context for the next pattern. If you are currently 'adding structure' to statement sequence, for example, the pattern language tells you that there are only four patterns that you need to consider when deciding on the next pattern - all the others can be ignored. This focusing of attention on the relevant patterns is especially important in the case of novices.

The third advantage is the converse of the first. The power of the pattern is itself twofold. Firstly it enables the details of the part of the problem that it solves to be effectively ignored while the rest of the problem is tackled. Then, once all the patterns are identified and the overall design of the solution attained, it enables the details that were ignored in the earlier stages to now be filled in easily, because they too are part of the pattern. Thus a pattern consists of two main functions or forces. The outer shell, the name of the pattern, functions as a component in the process of building up an overall solution, while the material encapsulated in the pattern enables the component details to be filled in once the overall solution has emerged. In other words, time is not wasted on fine details until it is clear that the overall design does indeed solve the problem. Once the 'pattern sequence' is complete the code example in each pattern is used to write the coded solution.

5 Conclusion

This is a powerful pedagogy. Each pattern is a small unit of knowledge, containing within it the means for the first four of Bloom's cognitive processes, knowledge, comprehension, application, and analysis. The

pattern language structure adds weight to the fourth factor, analysis, and provides the means for the last two, synthesis and evaluation, through the contextual information and the creative process that it provides. Patterns lend themselves to the learning of a skill, like programming, because they provide the static knowledge plus the means of applying it, the dynamics of language. In fact patterns have been criticised for being just a teaching aid. But the fact that it can be said that “when designers become experts they discard patterns” (Mattson 1996) and that “patterns are limited because of what they are - a teaching tool” (Booch 1999) seems to miss the point that this is true of all expertise. Once a technique is integrated into an expert’s practice its use occurs virtually unconsciously, like shifting gears in a car when driving. (Skemp 1971, p.55) It is criticising a pattern for being a pattern - the written software pattern has become a mental pattern. Of course, this is the thrust of this paper; the correspondence between the specialised meaning of the word in the design pattern sense and the normal meaning of the word in the mental pattern sense gives the concept its educational power.

6 Acknowledgements

An early version of this paper was workshopped at the third Asian Pacific Conference on Pattern Languages of Programs in May 2002. Jorge Ortega-Arjona, the participants in the workshop sessions at the conference, the members of the Flinders University KDM Research Group, and Jim Coplien all contributed valuable input.

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.
- Alexander, C. (2002), *Nature of Order*, Vol. 2, pre-press edn, Oxford University Press, New York.
- Bergin, J. (2002), ‘Coding at the lowest level, coding patterns for java beginners version 6’, At <http://csis.pace.edu/~bergin>. There are many links to other relevant material on Joe’s home page.
- Booch, G. (1999), ‘On patterns: straight talk on what makes patterns work’, <http://web2.infotrac.galegroup.com>.
- 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.
- Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st edn, Addison-Wesley, Reading, Mass.
- Mattson, S. (1996), *Object oriented frameworks: a Survey and Methodological Issues*, Lund University, Lund, Sweden.
- Skemp, R. (1971), *The Psychology of Learning Mathematics*, Penguin Books, Middlesex.

Sleeman, D. (1986), ‘The challenges of teaching computer programming’, *Communications of the ACM* **29**(9), 840–841.

Wallingford, E. (2001), ‘The elementary patterns home page’, At <http://www.cs.uni.edu/~wallingf/patterns/elementary/>.

7 Appendix

The Patterns used in the Example Problem.

Pattern Name: Program.

Problem: Designing a task so that it can be carried out by a computer.

Context: Designing a solution.

Forces: Automation, efficiency.

Solution: The program pattern is chosen if the problem specification suggests that applying a computer to the task of solving it is a useful or necessary thing to do. It sets the context for the problem to be solved by designing a program. In C a program is structured into three main elements. If a program uses some activity that is common to many situations, such as input and output routines, then it is likely that code for these routines already exist, and are incorporated in library files. If such resources are required, then these need to be incorporated in the program structure first, and in this case the `external resources` pattern is consulted. Any values that are required to be accessible throughout multiple functions in a program need to be handled next, and the `global variable` pattern covers this. As with the `external resources` pattern, if the need for such facilities is not obvious at this stage of development, then the pattern can be revisited later. The execution of a program is the third structural component, and it consists of a sequence of C statements partitioned into code blocks, called functions, on the basis of common purpose. The `functions` pattern describes this scenario. One of these functions is called `main` with the signature shown in the Code Example section of the `main` pattern. It indicates the starting point for the execution of the program, which is the process of carrying out the statements one by one until the end of `main` is reached. Some of the statements could involve calls to other functions, in which case their statement sequences are worked through. At the end of the statement sequence in a function the execution reverts to the point at which the function was called. The code is written in a file with the suffix `.c`.

Code Examples: N.A.

Related Patterns: N.A.

Pattern Name: External Resources.

Problem: Often designing a program involves performing very common activities.

Context: Program.

Forces: Efficiency.

Solution: The activities that are repeated in many programs have often been written into library files. Using facilities that exist in library files is just a matter of writing a reference to those library files that are appropriate to a particular program in the manner shown in Code Examples.

Code Examples: #include <libraryFile.h>

Related Patterns: N.A.

Pattern Name: Functions.

Problem: Fitting the program structure to the nature of the required task.

Context: Program.

Forces: Logical partitioning of task.

Solution: If analysis of a task shows that it can be logically seen as being composed of several subtasks, then the actions that make up each subtask should be grouped together in a function, and the functions called in 'main'. If it is not clear that logical groupings exist, then the actions are all written in 'main'.

Code Examples: N.A.

Related Patterns: Function, Main.

Pattern Name: Main.

Problem: Designing the activity of the program - the actions it needs to carry out.

Context: Functions.

Forces: The need for a start and end point to program execution.

Solution: The 'main' function in C is the location of the sequence of statements representing individual actions that compose the total functioning of the program.

Code Examples: Every program will have a main function as a starting point for its execution. It can optionally return an int value for indicating successful or unsuccessful execution and can take an array of null terminated char arrays, representing command line arguments, as a parameter.

```
main() {  
    <code>  
}
```

A main that returns a value (an int).

```
int main() {  
    <code>  
    return n;  
}
```

A main that takes command line arguments.

```
main(int argc, char *argv[]) {  
    <code>  
}
```

Related Patterns: Local variable, Function.

Pattern Name: Statement Sequence.

Problem: Designing the activity of a program.

Context: Function or main

Forces: The need to specify the activity in terms of the syntax of C.

Solution: The solution is to place the C statements that specify each step of the activity in a code block delimited by braces within the definition of a function.

Code Examples:

```
int square(int num){  
    num = num * num;  
    return num;  
}
```

Related Patterns: N.A.

Pattern Name: Repetition.

Problem: Repeating activity.

Context: Statement Sequence.

Forces: The need for actions to be repeated.

Solution: The solution is to separate the activity into a code block of its own, so that this block can be executed as many times as required. Of course, code can be repeated simply by retyping it several times if both the number of statements that need to be repeated and the number of times they need to be repeated is both known and small. In this case, retyping is the simplest option and the statements will not need to be separated into their own block. However if the number of repeats is either unknown or large, then a separate block for the repeated code will be necessary. There are two ways of putting code into repeatable blocks, the loop (see Loop) and the subprogram (see Subprogram). The subprogram solution requires a call to the subprogram at every repeat and this is ideal if the repeats are not contiguous. Otherwise the loop is probably the best choice.

Code Examples: N.A.

Related Patterns: Loop, Subprogram.

Pattern Name: Loop

Problem: You need to perform the same series of actions several times in succession.

Context: Repetition.

Forces: If the number of repetitions is small and fixed, it may be easier to simply type the instructions the required number of times.

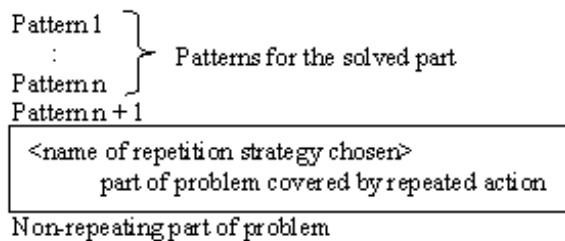
If the number of repetitions is large, retyping is inconvenient or impractical. Furthermore, if you ever need to change the actions to repeat, you will have to change all the copies.

If you don't know how many repetitions will be

needed, or if the number will vary as the program executes, you can't just copy the code because you don't know how many copies to create. If the series of actions to repeat is long, consider using a subprogram to group the actions together.

Solution: The general solution is to use a loop, which causes the flow of control to go back to the beginning of the block of instructions so that the instructions are re-executed. Several different kinds of loop are common, each using a different strategy to determine whether to repeat the body another time or not.

The loop pattern relates to the overall problem situation by dividing it into two separate components, that part of the problem that is dealt with by the repeated actions, and the part of the problem that isn't. This means that a repeated code pattern is best represented by boxing the repeating part, which will eventually be represented by a pattern or pattern sequence, away from the rest of the problem (the non-repeating component). This representation is illustrated below.



This creation of a separate block of code means that the pattern, statement sequence, is revisited to build the pattern sequence that solves the repeating part of the problem. This is the meaning of the upward pointing arrow. Once the repeating part is fully specified, the process reverts to the original line of development. The box in the figure illustrates this separation of the two sequences.

There are four kinds of loops to cover different situations.

The test-at-the-start strategy tests a condition before executing the loop body.

The test-at-the-end strategy tests a condition after executing the loop body.

The index-loop strategy executes the body a fixed number of times, setting a variable, the loop control variable, to successive values from a specified sequence.

The counted-repetition strategy, a simple version of the index-loop strategy, simply repeats the body a specified number of times.

Code Examples: N.A.

Related Patterns: Subprogram, While Loop, Do Loop, For Loop, Index Loop

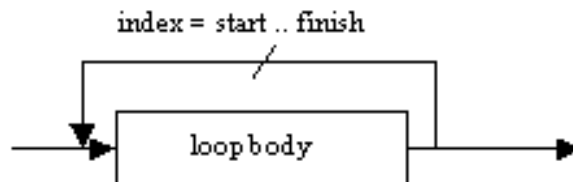
Pattern Name: Index Loop

Problem: You need to perform the same series of actions several times in succession, but using a different value each time.

Context: Loop.

Forces: Repetition with variation.

Solution: Index variable loops are appropriate when the actions in the body need to vary and the variation can be expressed as a series of values. The index-loop strategy, represented by the for loop, executes the body a fixed number of times, setting a variable, the loop control variable, to successive values from a specified sequence. Usually the sequence is specified by giving a starting value, an ending value, and an increment.



```
for(each value in a sequence)
  <do this>
then continue
```

Code Examples:

```
for(count = 10: count > 0; count--) {
  printf("%d ``green bottles``, count);
}
```

Related Patterns: Subprogram, While Loop, Do Loop.