Difficulties in Designing with an Object-Oriented Language: An Empirical Study[1]

Françoise Détienne

Projet de Psychologie Ergonomique pour l'Informatique
INRIA
Domaine de Voluceau,
Rocquencourt, BP 105
78153, Le Chesnay Cedex
France
Email address: detienne@psycho.inria.fr
Phone Number: (1) 39 63 55 22

Abstract

An experiment has been conducted to study the activity of program design developed by programmers experienced in classical procedural languages as they use an object-oriented language. We collected data on the design activity exhibited by eight programmers using an object-oriented language for solving two problems. Half of the subjects were beginners in object-oriented programming (OOP) and the other half were experienced in OOP. This paper focuses on the analysis of the difficulties programmers experienced in understanding and using the concepts and constructs of an object-oriented language. Our results underline the importance of transfer and reuse of solutions in the activity of design. .

1. Theoretical framework and goals

An experiment has been conducted to study the activity of program design developed by programmers experienced in classical procedural languages as they use an object-oriented language. We assume that experienced programmers possess in memory numerous schemas they have constructed through practice in their domain of expertise. Evidences supporting this hypothesis have been found in various studies (Détienne, in press; Détienne & Soloway, 1988; Soloway & Ehrlich, 1984).

It has been shown that experienced programmers possess schemas dependent on the task domain and schemas dependent on the programming domain. These schemas are evoked and used whenever programmers perform programming tasks. It is likely that they are more or less dependent on programming languages (specifically the type of language such as declarative, procedural, object-oriented) and on methodologies of design such as top-down design or relational design.

The originality of this study is double.

First, there is, as far as we know, no empirical study on object-oriented programming. Most psychological studies on software design (Adelson & Soloway, 1984; Visser, 1987) were conducted with programmers using procedural languages or, more recently, declarative languages.

An empirical study of programming with an object-oriented language seems particularly interesting to evaluate the claims made on the easiness to program with this kind of languages. They are assumed to make the design process easier inasmuch as it is natural to decompose a problem into objects and actions to operate on these objects. According to Meyer (1988) the world can be naturally structured in terms of objects, thus, it seems

[1] This research was supported by the GIP ALTAIR. Altaïr is a consortium funded by IN2 (a Siemens Subsidiary), INRIA (Institut National de Recherche en Informatique et Automatique) and LRI (Laboratoire de Recherche en Informatique, Université Paris XI).

particularly relevant to organize a model of design around a software representation of these objects.

Second, most psychological studies on learning programming were focused on knowledge acquisition by novices. Inasmuch as experienced programmers tend to have, more and more, new languages at their disposal, the learning of new programming languages by experts seems an important psychological question to address.

A question this study addresses is how subjects which are experienced in programming learn new concepts in their domain of expertise. We expect experienced programmers to use the schemas they have constructed in the programming and the task domains so as to construct new solutions more adapted to the new language which is, in our study, an object-oriented language.

However it is likely that having programming schemas at their disposal may have positive effects as well as negative effects. If the best model of design (as assumed by computer scientists) for the object-oriented language is closer to a model of the task domain than to models already constructed for familiar programming languages already known, then trying to apply programming schemas already constructed may produce negative transfer. In this case, we expect experienced programmers to encounter difficulties and to construct non adequate solutions.

In our experiment, we collected data on the activity of program design with an object-oriented data base system, the "O_2 System" (Bancilhon et al. 1988; Lécluse & Richard, 1989) which is being developed by the GIP ALTAIR.

2. Methodology

2.1. Subjects

Eight professional programmers participated in this experiment. All had several years of programming practice with classical procedural languages such as C, Cobol, Basic. Four of them, "beginners" in object-oriented programming (OOP), had no practice in object-oriented programming. The four others, "experienced" in OOP, had several weeks of practice with the object-oriented language under study.

2.2. Material

Inasmuch as the kind of problem to program has an effect on the strategies developed, and a strategy may be more or less easy to use in an object-oriented environment, we chose to give subjects two different problems: one problem is "declarative" inasmuch as it has been shown, in a previous study (Hoc, 1983), that the data structure guides the program development, the other problem is "procedural" inasmuch as the structure of the procedure has been shown to guide the program development. In the paradigm of object-oriented programming, identifying objects and their characteristics is important in the design process and in a declarative problem, these aspects can be more obvious to analyze. So we assume that an object-oriented language makes the programming activity, at least in a learning phase, easier for a declarative problem than for a procedural problem.

The two problems were problems of management (financial management and data base management). This task domain was familiar to the subjects.

2.3. Procedure

Each subject had the two problems to program. The order of problems presentation was counter-balanced. The programmers not experienced in object-oriented programming had one day for programming each problem whereas the programmers "experienced" in

object-oriented programming had half a day (which has been proved to be sufficient to develop the program at least as much as what beginners did). Previously to the phase of program design the four subjects "beginners in OOP", received a one-day theoretical formation. Subjects were asked to verbalize while designing their programs. They were allowed to ask questions to several experts in object-oriented programming whenever they had problems they were not able to overcome.

All subjects had at their disposal a manual for the system, a theoretical paper on object-oriented programming and an example-program i.e., a program written in CO_2 (the object-oriented language under study), solving a problem different from the experimental ones. After the programming phase, subjects had to answer questions on the difficulties they had experienced during the experiment.

We collected the subjects' verbalizations, successive versions of programs under development, notes written during the realization of the task, questions asked to the experts, the order in which the different traces of the activity were made, i.e., the order for writing notes and coding programs with the verbalization recorded simultaneously.

The final versions of programs have been given for evaluation to experts in object-oriented programming. They were asked to detect and report errors as well as "inelegances" in design and style. They had to rank them by order of seriousness, to classify them and to make explicit their criteria of classification.

2.4. The 02 System

The O_2 system is an object-oriented data base system. A "classical" language, is used mainly to write the methods. In the version of the system used for our experiment, this is language C, so the whole system is called "CO_2 system". An object-oriented layer, the CO_2 language, is added to the "classical" language. Subjects have been chosen so as to be familiar with language C. Thus they had to learn mainly the CO_2 language.

The object-oriented programming paradigm is based on the concepts of class, inheritance, message passing, late-binding. A class is defined as a structure (a type) and methods. A method is a function attached to a class that describes one part of the behavior of the objects which are instances of this class. A value is encapsulated in an object.

There are various possible relations between classes. The "is-a" relation defines a specialization between a class and its superclass. The "is-part-of" relation defines an imbrication between classes. A class inherits the properties of its superclasses. This inheritance property apply on structural properties of classes and on functional properties of classes, i.e., a class inherits the structure of its superclass and the methods associated to it.

A call to a method is termed passing a message. A subclass can redefine methods thus a method can have the same name and be associated to different classes with different code. When the program is executed, one of these methods will be called according to the class of the object on which this method is applied: this is the principle of late-binding.

A program has several parts. One part is the "model of classes" ("schema" in the terminology of O_2 system designers), i.e. the definition of classes which is composed of the type specification (the names of classes, the types and names of attributes, the relations between classes), and the method specification (the signatures of methods which are names and parameters of methods). Another part is the code of methods.

3. Results

Programmers developed fairly rapidly programs with the new programming language. The use of their previous knowledge in programming had a positive effect in acquisition of the new concepts of the object-oriented language. However, it is not possible to measure how easy it was to learn this new language compared to other languages.

Although designing with an object-oriented language seemed easy in some aspects, we will focus, in this paper, on the analysis of the difficulties encountered by programmers in performing their task. First, we present some characteristics of strategies used by subjects. Second we analyze some difficulties programmers experienced in using language-specific concepts. Then we analyze some reasons of programming difficulties which are mostly negative effects of transferred knowledge.

3.1. Characteristics of design strategies

The difficulty to program was judged different according to the type of problem solved. Beginners in OOP tended to judge the procedural problem more difficult than the declarative one: three over four beginners in OOP found designing the procedural problem more difficult than designing the declarative problem, the fourth one finding them equivalent. Two subjects "experienced" in OOP also found the procedural problem harder to program whereas the two others found the declarative problem harder to program.

The same kinds of reasons were given to compare the two problems by the beginners and the experienced in OOP. It is noteworthy that the declarative problem was judged more difficult for the structuration and composition of classes and for the association of the functionalities to classes; this is precisely what is assumed to be easy with this kind of language by the tenants of OOP. From the data on the design activity, it appears that programmers, specifically the beginners in OOP, experienced many difficulties for both kinds of problems. In a further analysis of our result we will evaluate whether or not some kinds of difficulties are problem specific.

The design strategies followed by the programmers have the following characteristics[2].

3.1.1. Anticipation of aspects of the solution

First the design activity consisted in identifying the types of classes and the functions (corresponding to individual methods) associated to them in a solution. Thus the process of class creation started previously to the writing of the bodies of methods. The subjects started their activity by the construction and the coding of the model of classes. We observed that classes were defined by programmers either in a top-down direction (from super-classes to sub-classes) or in a bottom-up direction. Novices in OOP had many difficulties to construct classes, to use the "is a" and "is-part-of" relationships between classes, and to associate methods to classes.

In this process of class creation, programmers tried to anticipate the different classes and methods they will need in the detailed coding of methods, i.e., the procedural aspects of the program. However, we observed they were not able to anticipate all these aspects. When the subjects judge the model of classes to be sufficient they started the coding of methods. Then the subjects made many modifications to the model of classes while writing the bodies of methods. These modifications are: addition of classes, addition of methods, addition of attributes in a class, addition of parameters in a method signature, modification of association between methods and classes (move a method

[2] A characteristic, we will not develop in this paper, is the use of simulation. We observed programmers simulating partial solutions developed at various levels of abstraction.

from one class to another).

3.1.2. Reuse of solution

We must insist on the importance of the reuse of solutions or parts of solution by both beginners and experienced subjects. They use transfer of knowledge in designing their solution, i.e., they evoke solutions or parts of solution, more or less abstract, that they try to reuse. These solutions come from internal source, i.e., the memory of the programmers, as well as from external sources, i.e., other programs (the example-program or programs written previously by the subject him/herself) or parts of the program being written.

As subjects develop their program, we observed that they evoke solutions already constructed either in the same language or in other languages. They often looked to the example-program written in CO_2 they had at their disposal in addition to the manual. Although this program solved a problem very different from the problems they had to deal with, they used it to extract examples of syntactic structures and semantic structures.

As soon as they have written a part of their program, they duplicate parts of programs using the copy command. For example, they duplicated parts of the model of classes so as to define new classes, using previous classes as templates. When they had methods performing the same functionality and being associated to different classes, they often duplicated the body of the first method written so as to use it as a template for writing the other methods.

Transfer of knowledge may have a positive effect as well as a negative effect. It has positive effects inasmuch as it allows subjects to use already constructed knowledge structures in their solutions without having to reconstruct them which would be more expensive. It has negative effects whenever the knowledge structure is used without being correctly adapted. In this case, transfer mechanisms cause errors. This is developed in the last section of this paper.

3.2. Difficulties with language-specific concepts

The programmers experienced the following difficulties in understanding and using concepts of the object-oriented programming language.

3.2.1. Assimilating a new concept to an old concept

When understanding a new concept, programmers try to relate this concept to already known programming concepts. Whereas this process helps the acquisition of new concepts, it can lead to misconceptions when programmers assimilate a new concept to an old one.

For example, we observed beginners in OOP have a misconception on what a class is. They confused the new concept of "class" and the already known concept of "set". A class represents a family whose objects structure is identical. They tended to conceive a class as a set of objects. This misconception led them to programming errors: they tended to associate first a method which processes a set of objects of class A to the class A itself instead of creating a class A' whose type is set(A).

3.2.3. Using a same concept in different contexts

Programmers had difficulties to use a same concept in different contexts. This was observed for the inheritance property. This property was more difficult to use when applied to functional properties compared to structural properties of classes.

The programmers beginners in OOP had less difficulties with the use of inheritance on structural properties than with the use of inheritance on functional properties. The programmers experienced with OOP had no difficulties in using the inheritance properties when applied to structural properties of objects. They even tended to construct situations in which it is possible to use this property in the future (if the problem evolves) although it is not necessary for the present solution.

For example, several experienced subjects used the type "tuple" (which is a record composed of parameters) as main type to describe structural aspects of classes even if there is only one parameter in the tuple. As most structures in database management have the type tuple, creating a class of this type allows to use the inheritance property as its type is likely to be compatible with any other classes type. This has been ranked as inelegant by evaluators whereas it is an overgeneralization of the way to use inheritance properties of structural aspects of the solution.

The inheritance properties on functionalities is difficult to use by both programmers beginners in OOP and experienced in OOP. From "inelegances" or "errors" detected by evaluators of the last version of the programs designed, we can say that programmers do not use this property each time they should so as to conform to OOP principles. They sometime reuse a known solution which had been constructed in programming with classical procedural language (without use of the inheritance property) instead of constructing a new solution which would use the inheritance property. Moreover, when they use the inheritance property, they often make errors. These errors may be linked to the lack of a good representation of control flow and data flow in their programs, in particular, when they use the late binding process.

Programmers also had difficulties in differentiating between concepts: they tended to confuse three kinds of entity, a class, an object and a value, when manipulating them in a program.

3.3. Negative effects of knowledge transfer in design

Many errors and "inelegances" have been detected by the evaluators in the final versions of the programs. From the analysis of the protocols collected on the design activity, we have been able to analyze how some of them are produced. It appears that the transfer of various kinds of knowledge has negative effects which explain many errors and "inelegances" in design. Subjects, beginners as well as experienced in OOP, transfer schematic structures they have constructed through their practice of programming. These transfers are useful in the learning process inasmuch as they produce a structure which can be adapted to a new device. However, in some cases, the programmers just apply the old structure without taking into account the new constraints and functionalities of the new device.

In this section we present examples of these negative effects of the transfer mechanism. We differentiate different cases according to the kind of schematic knowledge being transferred.

3.3.1. Schematic knowledge dependent on the task domain

Programmers possess schemas dependent on the task domain they evoke as soon as they have information on the problem they have to solve. From these schemas they can infer data structures and functions to perform for solving a certain kind of problem. For example, they know that, for data base management, which is the kind of problems they had to solve, there must be some kind of set of records in the program and some functions are to be performed: creation, modification, deletion.

According to the hypothesis on the naturalness of OOP, we would expect that schematic knowledge dependent on the task domain helps programmers so as to develop solution

adequate for OOP. Our data suggest that it is not the way it happens.

We observed that beginners in OOP experienced difficulties in structuring their solution. From the problem statements and from their schematic knowledge in the task domain, they inferred functions and objects to use in their solution. However, they found difficult to relate one to each other, so as to construct the model of classes, and often tried several solutions. The model of classes seems not to be transparent in the problem statements.

Furthermore, it appeared that very soon in the design process, the programmers, mostly the beginners in OOP, inferred elements of solution which do not conform to OOP principles. Although the representation they worked on was a very abstract one and still very close to the problem statements, they added elements of schematic knowledge relative to a methodology of design (different from OOP), or relative to a solution in classical procedural language. We develop below several examples of these mechanisms.

3.3.2. Schematic knowledge dependent on a methodology of design

Very early in their design activity, we observed that beginners in OOP may evoke and use schematic knowledge dependent on a methodology of design which do not conform to OOP principles. For example, a subject evoked elements of a solution constructed with a relational approach of data base management. According to this methodology, different objects have a number which is used as a cue to link together objects and to help the search in the data base. Evoking this schematic knowledge, the subject added an attribute of type "number" to each class he had constructed previously. Then he constructed a kind of "flat" structure of classes, without using the "is-part-of" relationship to link together classes. According to the evaluators, the final solution, which was a development of this abstract solution did not conform to principles of OOP.

3.3.3. Schematic knowledge dependent on classical procedural languages

Very early in their design activity, we observed that beginners, as well as experienced programmers in OOP, evoked schematic knowledge dependent on classical procedural languages. For example, a beginner added a parameter "type of object" in a class which allowed him to do different processing (method calls) according to the value taken by this parameter in a structure "case of" or "Ifs". This is typically a solution constructed for classical procedural language. In doing so, he does not take into account the functionalities of OOP. By the late-binding and inheritance properties, he could let the system decide during the execution which kind of object is under process so as to call the adequate method without using a "type" parameter and a structure of selection.

3.3.4. Schematic knowledge dependent on the OO language

In their learning process, subjects construct new structures which are dependent on the OO language and they try to transfer these structures in order to apply them in different situations. This is a learning process which sometimes causes errors by overgeneralization of the use of a structure.

For example, when they had several methods which performed the same functionality in different classes, they tried to use the inheritance properties. This is possible when the signature of the method is the same for the method associated to the superclass as for the methods associated to the subclasses. However, many errors were produced as they generalized this structure without taking care of the signature of methods.

4. Discussion

Programming with an object-oriented language is similar to programming with a procedural language relatively to the following characteristics: development of solutions at different level of abstractions, top-down and bottom-up development of solution, anticipation of aspects of the solution, importance of reuse of solution, use of simulation.

However, it appears from our study that designing a program with an object-oriented language is not so easy and so natural as the tenants of OOP say. For the declarative problem and the procedural problem, programmers experience many difficulties in understanding and using the new concepts and constructs of this type of language. This result underlines the need for a methodology of design in OOP and a programming environment which support, by particular characteristics, the activity of design.

In the learning process of a new language by experts, we have underlined the importance of transfer and reuse of solutions. We have seen that previous knowledge of programming languages may produce negative effects in the acquisition of the new language. Beginners as well as experienced programmers in object-oriented programming tend to not fully use the functionalities of the new programming paradigm. They sometime use inappropriately the syntax of the new language so as to translate an old solution.This result underlines the need for training the subjects with examples which take into consideration their previous knowledge and, in particular, the transfers they may do.

From questions programmers asked and from errors or inelegances made in programs, it appeared that beginners in OOP did not have a good representation of data flow and control flow in programs written in the object-oriented language. This should be taken into account in the training to the new language.

An object-oriented language could be more adequate to novices in programming compared to experienced programmers who need to unlearn, in some way, acquired schemas. Studies (Hoc, 1989) show the importance of knowledge transferred from the task domain in the learning processes exhibited by novices. Thus the claim on the learnability of object-oriented languages may be more relevant for novices (Rosson & Alpert, 1988).

This research has been conducted while the O_2 system is still under development. The designers of this system are interested in our observations on programmers' difficulties so as to take them into account for the further development of the language, for the training of the system's users and for the development of a methodology of design and the documentation.

References

Adelson, B. & Soloway, E. (1984) A model of Software Design. Research report 342, Yale University, New Haven.

Bancilhon, F., Barbedette, G., Benzaken, V., Delobel, C., Gamerman, S., Lecluze, C., Pfeffer, P., Richard, P. & Velez, F. (1988) The Design and Implementation of O_2, an Object-Oriented Data Base System. Technical report 20-88, GIP Altaïr, Rocquencourt.

Détienne, F. (in press) Program Understanding and Knowledge Organization: the Influence of Acquired Schemas. In P. Falzon (Ed): "Cognitive Ergonomics: understanding and learning Human-Computer Interaction", Academic Press, London.

Détienne, F. & Soloway, E. (1988) An Empirically-Derived Control Structure for the Process of Program Understanding. Research report 886, INRIA, Rocquencourt (to appear in International Journal of Man-Machine Studies.

special issue on "Psychology of Programming")

Hoc, J-M. (1983) Une méthode de classification préalable des problèmes d'un domaine pour l'analyse des stratégies de résolution: la programmation informatique chez des professionnels. Le Travail Humain, 46 (3), 205-217.

Hoc, J-M (1989) Analysis of beginner's Problen Solving Strategies in Programming. In T.R.G. Green, S.J. Payne & G. Van der Veer (Eds): The Psychology of Computer Use. Academic Press, London, 143-158.

Lecluse, C. & Richard, P. (1989) The O_2 Database Programming Language. Proceedings of International Conference on Very Large Data Bases. Amsterdam, 26 Août 1989.

Meyer, B. (1988) Object-Oriented Software Construction. Prentice Hall, International Series in Computer Science.

Rosson, M. B. & Alpert, S. R. (1988) The cognitive Consequences of Object-Oriented Design. Research report, RC 14191, IBM, N.Y.

Soloway, E. & Ehrlich, K. (1984) Empirical Studies of Programming Knowledge. IEEE Transactions on Software Engineering, SE-10 (5), 1984, 595-609.

Visser, W. (1987) Strategies in Programming Programmable Controllers: A Field Study on a Professional Programmers. In G. Olson, S. Sheppard & E. Soloway (Eds): Empirical Studies of Programmers: second workshop. Ablex Publishing Corporation: NJ, 217-230.

WHAT ARE THE "CARRY OVER EFFECTS" IN CHANGING FROM A PROCEDURALTO A DECLARATIVE APPROACH?

Jawed Siddiqi,
Dept. of Computer Studies,
Sheffield City Polytechnic,
Pond Street,
Sheffield, UK.

Babak Khazaei,
SCIT
The Polytechnic
Wulfruna Street,
Wolverhampton, UK.

Abstract

This paper highlights the carry over effects in changing from a procedural to a declarative approach. The results of a case study into programming in Prolog for a relatively simple problem is reported. The paper describes the different methods of solutions that these subjects used to solve the problem and argues that they can be explained on the bases of strategies used for problem decomposition and the choice of data representation. It argues that the methods of solutions used suffer from a "carry over effect" based on a procedural approach. In particular, that the choice of data representation used appears to be more important than the paradigm used.

1. Introduction

Programming in a logic based paradigm has gathered momentum in recent years. This is because, among several other reasons, that the use of predicate logic allows one to state a programming solution in a declarative form, and it is argued that this is more natural than a procedural form for a large number of problems [1]. Some cognitive scientists [2] have rightly argued on the issue of naturalness of declarative forms.

From a human factors point of view the problem of "PD-programmers" (i.e. those traditionally trained and experienced in a procedural approach) learning Prolog programming is twofold. On the one hand, they are required to express their solutions in a logic paradigm which is a novel idea because they are used to "procedural thinking". On the other hand, they would need to know and consider the "control flow" of a logic based language which may or may not be identical to procedural features they are familiar with. This combination in some cases can be confusing. At present there is little empirical evidence reporting this phenomenon.

One study reports that programmers who have been trained in and used programming principles based on the procedural style have difficulties in adapting to the declarative style [3]. We believe this is because these programmers seem to continue to use the principles of the former rather than the latter style. It is not unreasonable

to expect this because it is known that people have strong tendency to apply known methods rather than learn new methods. Therefore, we argue that for Prolog programming the underlying execution mechanism used by PD-programmer relies heavily on procedural/operational "thinking". This tendency produces what we call "carry over effects" which in certain circumstances can lead to misconceptions. There is an absence of published empirical evidence which elaborates on these carry over effects. The aim of our investigation is to provide an insight into the crucial issues that need attention in order to ease the transition of PD-programmers from a procedural style to a declarative style of programming. In so doing we will highlight the dual procedural and declarative models used by PD-programmers.

Section 2 discusses the declarative and procedural issues in Prolog, section 3 details the specifics of a case study and the results of this are presented in section 4.

2. Declarative and Procedural Issues in Prolog

Prolog is considered a declarative programming language. This is claimed to be advantageous because declarative aspects of programs are usually easier to understand [1]. However, we postulate that there are two ways of viewing a Prolog program, declaratively as a collection of relations or procedurally as an ordering of relations. To understand the difficulties that novice PD-programmers experience, we briefly investigate declarative and procedural aspects of Prolog that influences the process of program construction and understanding

A Prolog program consists of clauses which are either facts or rules. Facts are unconditional clauses that define relations between objects. Rules are conditional clauses that are executed if the head clause matches the data that occurs in its environment. A simple Prolog program to illustrate this is shown below:

```
parent(tom, jane).
parent(helen,jane).
parent(steve,helen).
grandparent(X,Y):-
        parent(X,Z),
        parent(Z,Y).
```

The first three clauses are facts about the parent relationship. Each states that the first argument is the parent of the second argument (e.g. tom is jane's parent). The fouth clause is a rule and we can see that it defines a grandparent relationship in terms of a parent relationship. For problems involving relationships Prolog appears to provide an attractive medium to express problem assumptions and to represent problem solutions. For example, in relational database applications Prolog can act as a "higher level language" that frees us from having to express (redundant) procedural details. However, in practice for most applications writing a Prolog program entails the

PD-programmmer employing an execution mechanism based on ordering of relations (a procedural view). For instance, in our simple example, if we introduce a new relation 'pred' which finds the predecessor of a person one possible solution is to write:

```
pred(X,Z):-
        pred(X,Y),
        parent(Y,Z).

pred(X,Z):-
        parent(X,Z).
```

This definition is declaratively sound, but the Prolog system is not be able to find a predecessor for 'jane' because Prolog executes the facts and rules in their particular order from top to bottom and therefore 'pred' recursively calls itself and the terminating condition for the recursion is never reached. We need to change the order of clauses to enable Prolog to succeed in finding predecessors to:

```
pred(X,Z):-
        parent(X,Z).

pred(X,Z):-
        parent(Y,Z),
        pred(X,Y).
```

The two versions illustrate that in order to write and understand a Prolog program, a programmer needs to view the program as a collection of facts and rules as well as a process of goal satisfactions. To investigate such issues a case study was carried out and an account of the preliminary results is provided next.

3. Case Study

32 second year under-graduate computer science students undertook an assessment for a one semester module on functional and logic programming. The students for nearly two academic years, had received training in and used a procedural approach to programming. The exercise was to produce a Prolog program for the "Bridge Hand Problem". The statement of the problem is as follows:

Write a Prolog program which accepts as input a representation of a bridge hand consisting of 13 cards supplied in random order. The program is required to produce as output:

(a) the hand of cards arranged in descending order by rank within each suit.
(b) the points value of the hand (counting 4, 3, 2 ,1 for Ace, King, Queen and Jack resp.)

An example output is as follows:

CLUBS K 10 9
DIAMONDS J 9 4 3
HEARTS A Q 10 8 2
SPADES 7

POINTS VALUE = IO.

The Bridge Hand problem was the subject of a previous observational study into designer behaviour involving programmers using a procedural approach [4]. The choice of problem was therfore well suited for an initial comparative study between procedural and declarative paradigms.

Although the majority of the students had difficulties in providing a complete working solution to this problem, sixteen of them succeeded in producing comprehensive working programs The analysis carried out were similar to that of Siddiqi [5] that is the solutions were compared to identify distinct approaches. The classification chosen was in terms of decision made concerning "the choice of representation". This led to subjects atttempts being classified into two solution types. One in which the subjects chose to transform the input representation to the desired output representation (i.e. an ordered set of values) by means of an explicit sort routine, hereafter referred to as *transform type.* The method of solutions involves splitting the hand into four newly created lists according to suits. Each card in the hand is inserted into the appropriate list according to its value.

The other in which subjects chose to process the input representation in its original form with the honour cards being revalued so as to facilitate the use of the In-built sort routine. This solution, hereafter referred to as *patch it type*, involves using a "patching" routine to convert the sorted list into the desired output. In terms of Siddiqi's previous work [5] *transform type* represents a "data driven" approach, because the primary focus is on processing the data stream. Whilst the *patch it type* represent s a "goal driven" approach, because the goal is to "sort" the hand using the built in sort routine.

From the 32 attempted solutions 24 (75%) were of the patch it type. The most likely explanation for this is that subjects were attempting to use a "do what you can and make the rest fit around it". A strategy reported by Siddiqi in the study of subjects using a procedural approach [5]. For the Prolog solution, subjects recognised the benefits of making use of the in-built sort routine (i.e. an island of certainty) and adding "patches" to facilitate this (fitting the rest around the island). It is hypothesised that the students who provided the transform type solution had used a data driven approach and did not rely on the built-in sort routine.