3-24-1993

Difficulties Experienced Procedural Programmers Encounter When Transferring to an Object-oriented Programming Paradigm

Scott Andrew MacHaffie
Portland State University

THESIS APPROVAL

The abstract and thesis of Scott Andrew MacHaffie for the Master of Science in Computer Science were presented March 24, 1993, and accepted by the thesis committee and the department.

COMMITTEE APPROVALS:

[redacted]
Jean Scholtz, Chair

[redacted]
Richard Hamlet

[redacted]
Sergio Antoy

[redacted]
Marjorie Terdal
Representative of the Office of Graduate Studies

DEPARTMENT APPROVAL:

[redacted]
Warren Harrison, Chair
Department of Computer Science

* *

ACCEPTED FOR PORTLAND STATE UNIVERSITY BY THE LIBRARY

[redacted] on February 8, 1994

ABSTRACT

An abstract of the thesis of Scott Andrew MacHaffie for the Master of Science in Computer Science presented March 24, 1993.

Title: Difficulties Experienced Procedural Programmers Encounter When Transferring to an Object-oriented Programming Paradigm.

Experienced procedural programmers seem to have difficulty when transferring from a procedural language to an object-oriented language. The problem is how to assist the experienced procedural programmers to make this shift. The long term goal of this research is to identify areas where programmers have problems and to develop an automated system to help them overcome these difficulties.

This study examines the class designs produced by procedural programmers and the effect of specifications and domain knowledge on class designs. Two types of specifications were used: those written from a procedural point of view which emphasized the functions and those written from an object-oriented view which highlights the domain entities. In addition, the problem specifications were selected from three different domains in order to assess the effect of domain familiarity.

Data was collected using paper and pencil designs and through verbal protocols. The class designs were analyzed to see if the different types produced could be classified and to determine the effect of specification type and domain knowledge.

DIFFICULTIES EXPERIENCED PROCEDURAL PROGRAMMERS ENCOUNTER WHEN TRANSFERRING TO AN OBJECT-ORIENTED PROGRAMMING PARADIGM

by

SCOTT ANDREW MACHAFFIE

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
COMPUTER SCIENCE

Portland State University
1993

Table of Contents

List of Tables

CHAPTER I

INTRODUCTION

Learning a new programming language causes difficulties for experienced programmers and university students. As the differences between the new programming language and the old increase, so do the problems encountered by the programmer.

A new programming paradigm, object-oriented programming, is exacerbating these problems. This paradigm is a radical departure from the procedural paradigm in which many programmers have been trained. Although experienced in procedural programming, they have trouble when shifting to object-oriented programming as this approach calls for an entirely different way of thinking about a solution.

PROBLEM STATEMENT

Experienced programmers are by no means reduced to the level of novices when they encounter a new language. However, significant difficulties still exist [1, 2]. I am interested in tools which will assist programmers in making this transfer. In order to create these tools, I must first identify the types of problems that programmers experience when making this transfer. I am investigating the problems encountered when procedural programmers transfer to an object-oriented paradigm. Specifically, what are the problems which experienced procedural designers have in designing object-oriented class hierarchies and can these difficulties be classified?

This thesis studies the difficulties experienced procedural programmers encounter when transferring to an object-oriented programming language and the effects of specifica-

tion type and problem-domain knowledge on these difficulties. Specifically, I am studying the design of classes and class hierarchies.

WHAT IS OBJECT-ORIENTED PROGRAMMING?

Object-oriented languages were designed for solving classes of similar problems [4]. That is, problems from the same domain. Object-oriented programming involves creating generic solutions to problems and then specializing these solutions for a particular problem. This contrasts with procedural programming which is concerned more with what functions the system or user needs to accomplish. Object-oriented programming also includes objects, which are members of a specific class. An object provides a set of functions which other objects can use to communicate with the object. A class is the definition for a set of objects. Classes can inherit functionality from other classes by being part of a class hierarchy. Objects in an object-oriented program exhibit encapsulation. This means that the internal details of an object are not visible to other objects.

GOOD OBJECT-ORIENTED DESIGNS

A good object-oriented design has several features. It uses objects to model the real world [5]. It incorporates encapsulation and data-hiding. It has class names which reflect meaningful elements in the problem domain; these are terms that can be used naturally in discussions of both the problem domain and the program [6]. Also, the naming scheme should reflect the terminology used in the actual problem domain [5]. Classes named for elements of the problem domain allow programmers to communicate more directly, if not more accurately, with problem-domain experts. Mapping classes to real-world objects allows for a more robust design [7]. Such classes should not change much over the life of a program, though their details may change [5]. A good object-oriented design should be based in the problem domain. The process of designing classes involves finding problem objects and their

relationships with each other [8, 5]. This process improves communications between different members of a programming team because of the naturalness of the design [3]. That is, the design can be discussed in terms of real world objects rather than programming language entities.

Classes should also show good encapsulation. Classes should not depend too much on the internal details of other classes [8]. If they do, then they provide more complexity for the programmers to keep track of, which can lead to maintenance problems or a failure to communicate between programmers [9]. This also leads to hidden dependencies which increases the cognitive demands of the system [4]. Hidden dependencies are links between parts of a program which are not explicitly written in the code. Because the links are not visible in the program directly, the programmer has more details to remember. This is what increases the programmer's cognitive demands. Also, better encapsulation results in a lower coupling between objects, which results in a better design [10].

The depth of the inheritance tree is also a measure of the "goodness" of an object-oriented design [10]. The deeper the inheritance tree, the more code is being reused within a class. Related to this is the number of children per class [10]. The more children a class has, the more the code in that class is being reused.

APPROACHES TO OBJECT-ORIENTED DESIGN

CRC

How does a designer go about finding the objects to model in his solution? Several methods have been defined. One approach towards object-oriented design is CRC (Class, Responsibility, and Collaboration) cards [6]. This approach designs classes in their relationship to other classes. The design process involves using index cards which contain the class name, the responsibilities (or services provided) of the class, and any collaborators (or other

related classes). The use of physical cards is an important part of this method. Cards can be physically grouped to show groupings within the class hierarchy. Inheritance can be shown by putting cards "below" the classes they inherit from. The only recommendations this method makes for finding classes or objects is that the words used to describe the objects be meaningful in the problem domain [6].

This method is in agreement with the principles of good object-oriented design discussed earlier. It suggests finding objects and classes directly from the problem domain.

Object-Oriented Structured Analysis

Another approach to object-oriented design is an object-oriented version of structured analysis [11]. This is an extension of a real-time structured-design method. This adds entity-relationship diagrams to structured analysis. An entity-relationship diagram represents a problem-domain component and its relationships with other entities. One of the changes between the real-time structured analysis method and the object-oriented version is that the original method only included an entity if the system needed to store information about that entity. The object-oriented version includes all relevant entities whether or not the current system will use them. This approach includes "stimulus-response partitioning" which involves finding external stimuli which the system needs to respond to. These are represented in the analysis as "transformations". In the object-oriented version of structured analysis, finding objects or groups of objects takes precedence over these transformations. In fact, the highest levels of the transformation hierarchy are supposed to be created on this basis. These are combined by grouping related transformations (which are operations) into higher-level transformations which are objects. The method also suggests that objects should have simple interfaces to each other.

This method is in agreement with the principles of good object-oriented design described earlier. It suggests using problem-domain objects as objects in the program. It

recommends simple interfaces which is the same as having good encapsulation [8]. Simple interfaces imply that there is a small number of dependencies between classes.

TRANSFER

Transfer is the influence of past learning on future learning. When past learning facilitates future learning, transfer is positive. When past learning hinders future learning, transfer is negative. There are two types of transfer which take place: problem solving transfer and learning transfer. Problem solving transfer occurs when one has already constructed a solution to a given problem. Research has shown that this type of transfer results in a decrease in the time it takes to write a solution in one programming language after having written a solution in another language [12]. The assumption is that the programmer has knowledge of both languages. Katz [13] suggests that subjects develop knowledge of a solution which they are able to use when writing solutions in other languages. Wu and Anderson [12] found that problem solving transfer did not involve actual program code; the languages were different enough that the actual solutions could not be transferred. Thus, what was being transferred was at a higher level. A large portion of this transfer effect was found in a decrease in the time needed to produce the first draft of a solution in the second language. The number of drafts which subjects needed to obtain the correct solution decreased although the average time per draft remained about the same. The number of drafts roughly corresponded to the number of semantic errors the subjects made. Wu and Anderson observed that transfer in their work occurred mainly at the algorithmic and problem levels. That is, subjects often used highly similar algorithms in the two languages. Even when different algorithms were used, transfer was evident. Wu and Anderson believe this involves problem-related knowledge and understanding of the problem.

Learning transfer is transfer of the essentials of programming and programming languages. This is the transfer of the basic concepts of programming which do not need to

be relearned for each new language [15]. Positive transfer occurs when similarities exist between programming languages. Learning transfer is studied by observing programmers learning a new language and noting where difficulties are observed.

Scholtz and Wiedenbeck [2] found that the difficulties experienced programmers encountered differed depending on the degree of dissimilarity between the language being learned and the languages which the programmer knew. Similar languages (i.e. languages for which the same algorithms suffice) caused only syntactic and semantic difficulties. Distant languages caused problems at the planning level.

PLANS

Experts in a procedural language develop "plans" or "schemas" which they can use when learning a new procedural language [16, 17, 14]. A plan is a knowledge representation in which the essential features of a solution strategy have been abstracted [16]. Plans are used by both novices and experts, but experts use plans more than novices [17]. There are several levels of plans: strategic, tactical, and implementation. The differences are in terms of scope. A strategic plan is a high-level strategy of how to accomplish a goal. A tactical plan is a realization of a part of a strategic plan. An implementation plan is how to accomplish a tactical plan in a particular language. The strategic and tactical plans are considered language-independent. Implementation plans, by their nature, are language-dependent. An example of a strategic plan is a'"Read/Process" strategy. This involves reading a data value and then doing some kind of computation on it. An example of a tactical plan is a "counter-controlled running total loop" plan which describes how to accumulate a sum [16]. This plan involves using a counter to process a set of data while accumulating a running total. An example of an implementation plan is the "for loop" plan from Pascal which would be used to implement the previous tactical plan. When learning a new language, experts must acquire new implementation plans. However, even between

procedural languages there are problems with plan transfer. This occurs because tactical planning is not completely independent from implementation planning [2]. Tactical plans can be influenced by implementation details. For example, a programmer used to thinking in terms of strings will be unlikely to produce a solution in terms of character arrays. Thus, some tactical plans are actually language-dependent.

There is some evidence that knowledge of procedural programming languages hinders object-oriented design activities. For example, procedural programming knowledge leads to errors and non-object-oriented programming style in novice object-oriented programmers' programs [14]. That study examined four experienced object-oriented programmers and four beginners to object-oriented programming who were experienced procedural programmers, using the CO2 language. Although both the experienced and novice object-oriented programmers used plans, the beginners did not know which plans to use or used the wrong one [14].

Low-level (i.e. implementation) plans and objects are basically orthogonal [18]. An object can participate in more than one plan, and a plan can require multiple objects. I am investigating what happens when procedural programmers, who are used to using plans, try to encapsulate these plans into objects. Manns and Carlson [1] indicate that negative transfer will occur in this situation.

COMPARISONS BETWEEN PROCEDURAL AND OO PROGRAMMING

Some studies have been done on the difficulties encountered by procedural programmers transferring to an object-oriented programming languages. Detienne [14] has studied programmers switching from C to CO2, and Pennington [19] has studied programmers transferring from C to C++. These studies have confirmed the fact that procedural programmers have problems when using object-oriented languages. However, these papers have looked more at the process as opposed to the results. They have examined the process of design

more than the use of any particular design strategy. The papers talk about the subjects' difficulties in general but do not classify the design strategies used. Detienne [14] observed that some of the behavior exhibited by the procedural programmers was enforced by the programming language being used. This particular language, CO2, required a class to be fully defined before it could be used. The major problems with these studies are twofold: they only used a small number of subjects and they have not categorized the solutions produced. This study addresses both of these problems. The studies conducted in this thesis used a larger number of subjects (12 subjects, 15 subjects, and 35 subjects) than any of the other studies, which averaged fewer than 10 subjects. This thesis also examines in detail the types of solutions novices produced and classifies the design strategies exhibited by these solutions.

DIFFICULTIES IN OBJECT-ORIENTED PROGRAMMING

Object-oriented programming's proponents claim it is a better paradigm than procedural programming. The benefits usually claimed for object-oriented programming include the naturalness of object-oriented programming and the reusability of object-oriented programs. The claims about reusability will not be examined in this thesis.

Researchers have cast doubt on the naturalness claim [8]. The naturalness claim will be examined to some extent in this study. Novice object-oriented programmers' designs will be examined and compared to a good object-oriented design (which presumably is "natural"). Some problems do not map well to the object-oriented domain, such as problem domains in which problems or solutions do not share similar features. Another problem is that programmers who are not familiar with the object-oriented language they are using may not produce designs which are optimal in an object-oriented sense

Object-oriented programming requires radically different approaches than procedural programming [1]. Object-oriented programming encourages solution in the problem domain

rather than being constrained by the computer [8]. The different approach required for object-oriented programming has implications for teaching. For example, class hierarchies must be introduced carefully to avoid causing misunderstandings for procedural programmers [1].

Designing the class hierarchy in an object-oriented language can be difficult. Even for experienced object-oriented programmers, there can be too much complexity. Programmers cannot manage the number of classes and the relationships between them by themselves. They need to have a good understanding of the problem domain and some kind of tool to help manage the complexity of the design [3]. Specific domain knowledge can reduce the complexity of this task. Dvorak and Moher [3] studied two different problem domains: vehicles and dinosaurs. The researchers determined through interviews that none of their subjects were familiar with the dinosaur domain. The experimenters assumed that the subjects were familiar with the vehicle domain. The results showed that the subjects produced designs that were in closer agreement in the vehicle domain than in the dinosaur domain. The subjects also used less time to produce the designs in the familiar domain than in the unfamiliar.

TOOLS IN OBJECT-ORIENTED PROGRAMMING

There have been some tools designed or proposed to help with the difficulties in object-oriented programming. A cognitive browser which would help with object-oriented programming has been envisioned [4]. The major conclusions of of these researchers is that a good browser improves the ways that programmers can get information about the objects and classes in a program. Green, et al. [4] also concluded that having multiple views of the code would be helpful. They suggested that the way to design a useful tool would be to figure out what cognitive problems programmers are having with current object-oriented programming environments and try to fix those problems. They envisioned a browser which

would let programmers create their own groupings, similar to the Smalltalk class hierarchies, which could be used to view or scan classes.

Another tool which has been constructed is a tutoring system for Smalltalk [20]. As with the cognitive browser, this study examined psychological issues among programmers in an effort to improve the design of tools. This tool was designed to assist in the transfer from procedural programming languages to object-oriented languages, among other things. Helping to manage goals, evaluating goals, filtering the class hierarchies, comprehending code, and critiquing the design were the other design goals in developing the tutor. The critique of the design gave the subjects feedback on how their solutions could be improved, even if the solutions were correct. However, these tools do not offer assistance in determining what should be objects or classes. They provide no assistance in critiquing a design in terms of how well the objects map into the given problem or in selecting design alternatives.

Fischer [21] has looked at domain-oriented design environments. These environments help the user explore design alternatives. This tool contains many example designs in limited domains. When a user enters a design, the system examines the designs it has on file and critiques the new design based on these examples. The system engages in an interactive dialogue with the user, allowing for exploration of the design space. Fischer's design environments are designed for end users who are nonprogrammers.

CHAPTER II

METHODOLOGY

This chapter describes the methodology used for the various empirical studies conducted for this thesis. This chapter also describes the conventions used in this thesis.

PAPER AND PENCIL

Several studies were conducted which involved pencil and paper (as opposed to computer) design. All of these studies received approval from the Human Subjects Review Committee. These studies involved undergraduate and graduate computer science students from both Portland State University and Oregon Graduate Institute. The data were collected from these students in two ways: from an examination given in an object-oriented programming class and from problem statements given to volunteers. The students were not paid for their participation in the examination. The examination lasted 75 minutes of which the data collected for this study comprised a large percentage. This data comprises the "train" problem domain. The rest of the students were volunteers who were paid $15.00 for completing two designs, one in the "post office" domain and one in the "insurance" domain. These volunteers were given two hours to complete two designs. They were also given two different types of specifications. One of these specifications was designed to highlight the problem-domain objects. This is referred to as the "object-oriented" or "OO" specification. The other specification was a more traditional type which highlighted the functions which the system and the user needed to perform. This is referred to as the "functional" specification.

All of these students were experienced procedural programmers who were taking an introductory object-oriented programming class. These students are referred to as "novices" in the study.

VERBAL PROTOCOLS

Studies were also conducted with experienced object-oriented designers. These designers were volunteers recruited from both industry and PSU. All of these designers had at least a year's worth of experience designing in an object-oriented environment. These subjects were paid $20.00 for a two-hour study. These subjects were asked to "think aloud" as they worked. They were videotaped to record their verbalizations. These protocols were then analyzed. In addition, the designs they produced were analyzed in the same manner as the pencil-and-paper designs described earlier. Verbal protocols were only run on the experienced object-oriented designers.

CONVENTIONS

Several conventions are used throughout this study. Class names are printed in italics to distinguish them from the real-world objects in the problem domains. For example, "ticket" refers to a real or problem-domain ticket, whereas "*ticket*" refers to a solution entity.

The appendices include the problem specifications given to the subjects as well as example solutions. The example solutions are idealized versions of the strategies and do not represent a single subject's design.

The solutions identified as "good" solutions in various locations are solutions which agreed with the "expected" solutions. The expected solutions are what I expected experienced object-oriented designers to come up with. I had no expectations about other types of solutions. The other solutions were identified after examining the data.

I was not looking for multiple inheritance in the solutions, although nothing in the problem statement prevented it.

CHAPTER III

TRAIN SYSTEM

This study involved two sets of subjects: novices and experts. There were 34 novices of which 33 finished the problem. Data was collected from the novices using a paper-and-pencil study. Two experienced object-oriented designers also produced designs for the train system. I collected paper and pencil designs and verbalizations from the experienced designers.

This problem involved simulating a train system in which a user could buy tickets or monthly passes and make or cancel reservations. The solutions were analyzed and classified according to the class hierarchy and relationships between the classes.

NOVICE ANALYSIS

The solutions displayed a variety of strategies. In fact, some of the novices exhibited more then one strategy in their solutions. The solutions are classified into six categories: good solutions, monolithic solutions, solutions exhibiting classification by problem-domain attribute, solutions exhibiting a modular breakdown, solutions exhibiting functional inheritance, and unique solutions. As mentioned in the methodology section, "good" solutions are solutions which matched the "expected" solutions. Table I lists the number of each type of solution.

TABLE I

SOLUTION STRATEGIES IN THE TRAIN DOMAIN

good solutions	4
monolithic	9
classification by attribute	9
modular breakdown	7
functional inheritance	11
unique solutions	1
total	41

GOOD SOLUTIONS

The distinguishing characteristic of good solutions is that functionality and attributes from the *ticket* and *pass* classes were abstracted into a more general superclass. Thus, the *ticket* and *pass* subclasses could inherit shared data and functionality while maintaining conceptual independence. Although four of these solutions contained this distinguishing characteristic, they also contained other nonoptimal characteristics.

One of the solutions may have been a direct result of the problem specification. The phrasing and the structure appeared to come directly from the table of prices. The names of the classes were *ticket* and *passes, monthly*. The responsibilities for the classes were: *1st class, reserved*; *1st class*; and *2nd class* for *ticket* and *1st class*, *2nd class*, and *reservation* for *passes*. The superclass for the *ticket* and *passes, monthly* classes was called *prices*.

Another solution failed to unify everything that could have been. This solution duplicated *firstClass*, *secondClass*, and *reservation* variables in both the *ticket* and the *pass* subclasses of the abstract *ticket* class. It would have been more logical to have put these variables into the abstract *ticket* class.

This behavior was also seen in another solution. This solution duplicated the function *make a reservation* in both the *ticket* and the *pass* classes even though this function could have been included in the abstract *ticket* class. This solution contained inconsistencies as there was a *cancel reservation* function in the abstract class, but the *make a reservation* function was at a lower level in the class hierarchy.

The other good solution multiplied classes unnecessarily and inconsistently. The *ticket* class had subclasses for first class and second class tickets, which increased the number of classes without real benefit. The *pass* class had no subclasses which was inconsistent with the *ticket* class. Also, the *first class ticket* class had a subclass for reservations, but it also had a *reserved* variable. The *reserved* variable was inconsistent with having a subclass for

reservations.

MONOLITHIC SOLUTIONS

The monolithic solutions were characterized by a single class containing all ticket and pass information. That is, a single class contained all the functionality and attributes for tickets and passes.

In this grouping were several subgroups. One subgroup (five of nine solutions) divided responsibility for handling reservations into a separate class from where the tickets themselves were handled. This subgroup will be called "monolithic/reservation". Another subgroup (three solutions) combined reservation and ticket information into the same class but also created a class which contained the prices for each different type of ticket and pass. This subgroup will be called "monolithic/prices". The other subgroup contained a single solution which put both reservation and ticket information into one class (the *ticket* class) and also had a class containing the number of reserved seats on each train. This subgroup will be called "monolithic/seats".

The solutions in the monolithic/reservation subgroup were consistent about putting all of the reservation functionality into the *reservation* class. There was no overlap with the *ticket* class. Some of the other solutions put functions such as *changeReservation* into the *ticket* class.

The solutions in the monolithic/price subgroup were very similar. The *price* classes contained instance variables named for each possible type of ticket or pass, such as *PricePassFirst* or *PricePassSecond*. The *ticket* classes were also similar, though they showed more variation than the *price* classes. The *ticket* classes contained data such as customer name and address. Two of the solutions contained variables such as *buyTicket* and *exchangeTicket*; this may have been a poor labeling on the subjects' part of what was a variable and what was a function.

A monolithic/seats subgroup consisted of one solution which divided the problem into a domain class (*customer*) and a solution class (*TrainBookings*). The *TrainBookings* class maintained a list of trains and how many seats each train had reserved. All of the other data and functions which would have been visible to the customer and which were in the specification were in the *customer* class.

CLASSIFICATION BY ATTRIBUTE

These solutions were characterized by a class for each possible type of ticket and pass: first class ticket, first class ticket with a reservation, second class ticket, first class pass, and second class pass. These solutions indicated a classification by problem-domain attribute. The problem domain had several important attributes: ticket or pass, first or second class, and reservation or not. Within this group were three subgroups based on the relationship between tickets and passes. One subgroup (three of nine solutions) had no relationship between tickets and passes. This subgroup will be called "class/modular". Another subgroup (four solutions) had passes as subclasses of the different ticket types (first class and second class). This subgroup will be called "class/inheritance". The last subgroup (two solutions) had all the different possibilities as subclasses directly under a *ticket* class. This will be called "class/multiplicity".

The solutions in the class/modular subgroup were very consistent about inheritance. All of the shared data went into the top-level class (i.e. the *ticket* or *pass* class). Variables and methods were not duplicated in the subclasses such as *first class ticket*. One solution took the subclassing to an extreme: in addition to creating subclasses based on the attributes given directly in the specification, this solution also created subclasses based on whether a ticket was one-way or round trip. Thus, under the ticket class were six subclass: *one way first class reserved*, *two way first class reserved*, etc. This solution showed a heavy reliance on domain attributes for class decomposition.

The solutions in the class/inheritance subgroup had different specific styles of inheritance, though they all showed general similarities. Two of the solutions had *FirstPass* and *SecondPass* inheriting from *FirstClass* and *SecondClass* tickets. All of the data and methods went into the *Ticket* superclass of *FirstClass* and *SecondClass*, except for the reservation data and methods, which went into the *FirstClass* class. One of the solutions exhibited inconsistent behavior in this case: the solution included reservation functions in the *FirstPass* class, even though the functions were inherited from the *FirstClass* class. This was inconsistent with the fact that no other functionality was duplicated in the solution. Another solution had a *ticket* class with *FirstClass* and *SecondClass* as subclasses. The unique part of this solution was that passes were found in a *MonthlyPass* class which was a subclass of *FirstClass*. The class of a pass was stored as a variable called *kindOfClass*. Presumably this was done to inherit the reservation functionality from the *FirstClass* class. No data or functions were duplicated in this solution. Another solution in this subgroup contained a *ticket* class. This class had subclasses *FirstClass* and *SecondClass*, but with a difference: these classes contained both tickets and passes. The type information (ticket or pass) was stored in a variable named *Type*. Interestingly enough, this variable was not inherited from the *Ticket* class, but appeared separately in both the *FirstClass* and the *SecondClass* classes. This also occurred with the function *addReservation*, although *changeReservation* and *cancelReservation* appeared in the *Ticket* class. There was also a *FirstClassReserved* class which was a subclass of *FirstClass*. Unfortunately, the subject ran out of time at this point and there were no details provided for this class. It is not clear why this class was needed, as reservation functions were provided by the *Ticket* superclass.

The other subgroup was the class/multiplicity subgroup which created a subclass under the *Ticket* class for each attribute. One of the solutions used the subclasses *FirstClassTicket*, *SecondClassTicket*, *FirstClassPass*, and *SecondClassPass*. The other solution combined *FirstClassPass* and *SecondClassPass* into one class called *MonthlyPassTicket*. This

solution also had *FirstClassTicket* and *MonthlyPassTicket* inheriting from a *Reservation* class. In the first solution, the differences between the classes were that all of them except for *SecondClassTicket* contained a reservation variable and functions for adding, changing, and cancelling a reservation. There was little encapsulation in this solution: there was a class called *reservation*, but the functions for dealing with reservations were stored in four separate classes, none of which was the *reservation* class. The other solution did not repeat any functionality–all data and methods appeared exactly once in the class hierarchy.

MODULAR BREAKDOWN

These solutions were characterized by having independent *ticket* and *pass* classes. These classes did not participate in any direct inheritance relationship, with the possible exception of being subclasses of *Object*, which is required in Smalltalk.

The most consistent feature of these solutions was duplication of data and methods. The *ticket* and *pass* classes contained data or methods which were the same but which had to be duplicated in each class. This problem would have been eliminated if the two classes had been unified by an abstract class, as was done in the "good" solutions. This duplication was also found in other places in these solutions. Two of the seven solutions duplicated reservation methods such as *add a reservation* or *change a reservation.* Sometimes code or functionality which should have been duplicated was not. One solution had *cancelReservations* in the *ticket* class but not in the *pass* class.

FUNCTIONAL INHERITANCE

These solutions were characterized by passes inheriting data and functionality from tickets, or vice versa. These solutions were divided into three subgroups: one subgroup had subclasses under the *ticket* class for the types of tickets available, such as *FirstClass* or *SecondClass.* This subgroup contained five of the eleven solutions. It will be referred to

as the "func/multiplicity" subgroup. Another subgroup contained five of the solutions and consisted of merely a *ticket* and a *pass* class with one being a subclass of the other. This subgroup will be referred to as the "func/direct" subgroup. The other subgroup consisted of a single solution which was similar to the previous subgroup except that it contained an additional subclass. This subgroup will be referred to as "func/unique".

The func/multiplicity subgroup was discussed above in the heading "Classification by attribute". The important features to note for this section are that four of the five solutions in this subgroup contained no repetition of data or methods. In these four solutions, every data item and method which did not need to be specialized for each class appeared exactly once in the class hierarchy. The fifth solution was unique in that it duplicated the reservation methods in both the *FirstClass* class and in its descendent, the *FirstPass* class.

The func/direct subgroup also contained no duplication of functionality or data. All of the data and methods occurred exactly once in the class hierarchy.

The func/unique subgroup was unique for this strategy (functional inheritance) in that it contained an additional subclass. The hierarchy for this solution was a *Ticket* class with subclasses *Pass* and *ReservedTicket*. Again, no functionality or data was duplicated inside the class hierarchy.

UNIQUE SOLUTION

This solution was characterized by explicitly procedural thinking. This solution contained classes such as *BuyTickets* and *CancelReservations* which are more correctly implemented as methods inside classes. These classes were subclasses of a *Transaction* class. As this hierarchy was not reflected in the problem specification, it must have been based on the subject's outside knowledge. This hierarchy also included a breakdown by attribute: there were classes entitled *FirstClassWithReservation*, *FirstClassWithoutReservation*, etc.

OTHER CLASSES

The following are some other features of the solutions broken down by solution type. Table II lists the way reservations were included in the system. The categories are reservation as a class, reservation as a member of the *ticket* class, and implicitly stored in the hierarchy. Some of the solutions stored reservations as part of the class hierarchy; these solutions included classes such as *FirstClassTicket* and *FirstClassTicketWithReservation*. This is storing the reservation information implicitly in the hierarchy.

TABLE II

RESERVATION INFORMATION IN THE TRAIN DOMAIN

	Reser. class	Reser. in ticket	implicit	total
good solutions	1	2	1	4
monolithic	4	5	0	9
class. by att.	4	5	0	9
modular	5	2	0	7
func. inherit.	4	6	1	11
unique	0	1	0	1
total	18	21	2	41

Including reservations as a separate class was a slight improvement over including reservations as part of the *ticket* class. Using a separate class for reservations would allow these classes to be generalized. Also, having *reservations* as a separate class allows other things besides tickets to have reservations made. For example, a car rental system could use reservations but would not need tickets. The solution of not explicitly including reservations is a poor choice. This makes the train system entirely dependent on the structure of the class hierarchy; there is an implicit dependency which is not explicit in the code but is only apparent after careful study of both the problem and the solution. This requires the programmer to maintain extra knowledge which may be forgotten or may not be passed on to new programmers. This creates maintenance problems. Also, the functions to make or

cancel a reservation have to explicitly know about both the "with" and "without" reservation classes, which violates the principle of encapsulation.

Table III indicates how many solutions included some kind of driving class, which might be a holdover from the concept of a "main" program. This class was usually named something like *TrainSystem* or *Station*. The inclusion of a driving class has no impact on the object-oriented quality of a solution. Few of our subjects used a driving class. This might have been higher if the subjects had implemented their solutions as well as designing them.

TABLE III

INCLUSION OF A DRIVING CLASS IN THE TRAIN DOMAIN

	driving class	no driving class	total
good solution	1	3	4
monolithic	2	7	9
class by att.	0	9	9
modular	0	7	7
func inherit.	0	11	11
unique	1	0	1
total	4	37	41

Table IV indicates how many solutions included a *customer* class. A good object-oriented design should include a *customer* class to model the user, but many subjects did not include such a class. This was brought out in the second paragraph of the specification, which begins "A customer can..."

TABLE IV

INCLUSION OF A CUSTOMER CLASS IN THE TRAIN DOMAIN

	customer class	no customer classtotal	
good solution	1	3	4
monolithic	3	6	9
class by att.	2	7	9
modular	3	4	7
func inherit.	4	7	11
unique	0	1	1
total	13	28	41

EXPERT ANALYSIS

In addition to the novice solutions, I collected two expert solutions to use as a basis for comparison. One of these subjects produced a design which exhibited both a modular strategy and a classification by attribute strategy. This solution used two independent abstract classes called *Ticket* and *Pass*. Both of these abstract classes had subclasses based on attribute (i.e. *FirstClassWithReservation*, *FirstClass*, etc.). According to a rejected design, the subject had also considered using *one-way* and *round-trip* as part of the class hierarchy. This solution did not duplicate functionality or data between the *ticket* and the *pass* classes, so this may be the explanation for why the subject did not come up with a "good" solution. This solution put the functionality which was in the abstract *ticket* class in the good solutions into a *Station* class. This class had functions for dealing with reservations and for buying tickets and passes, with or without reservations.

The other solution exhibited the expected "good" strategy. This solution had three classes: an abstract *Generic Ticket* class, a *Ticket* class, and a *Pass* class. This solution included data such as customer name and class type (first or second) in the abstract class. Neither functionality nor data were duplicated in the class hierarchy. This solution was in

agreement with the expected solution.

VERBAL PROTOCOLS

Verbal protocols were collected from the experienced object oriented designers in addition to pencil and paper designs. These protocols were analyzed to see what design decisions were made and what the reasoning was behind the decisions. Due to technical problems with the audio portion of the video tape, one protocol contained no useful data other than the fact that the subject was considering whether to combine the *ticket* and *pass* classes into an abstract class.

The protocol for the other subject showed that several design decisions were made. After reading the problem specification, this subject noted that tickets and passes looked like they would be objects. This subject considered making a subclass under tickets called *ticket with reservation* but rejected this idea on the basis that the reservation status could be changed dynamically. The subject then proceeded to consider whether or not to unify *tickets* and *passes* into an abstract class. At the same time, the subject was considering whether or not to make reservations into a class. The subject did not give any basis for a decision but proceeded to design a *Generic Ticket* class which was an abstract superclass for both *tickets* and *passes*. The subject proceeded in a top-down design manner, starting with the highest level class and proceeding to its descendents. The order of class creation was *Generic Ticket* followed by *Ticket* followed by *Passes*. After the design was finished, the subject went back to the specification to verify that the design met all the requirements of the problem. The subject explained how the classes he had designed could be used to satisfy the functionality required by the specification.

CHAPTER IV

POST OFFICE ANALYSIS

This problem included two groups of subjects: novices and experts. Paper and pencil designs were collected from the novices. Designs and verbalizations were collected from one experienced designer.

This problem asked the subjects to design a simple simulation of a post office. The simulation included letters (first class, second day, and overnight delivery), post cards, and packages. The subjects were given two types of specifications: functional and object-oriented. These specifications are in Appendix A.

NOVICE ANALYSIS

The solutions displayed a variety of strategies. All of the strategies found in the other problem domains are listed in the table below, although some of them were not evident in this study. Again, several subjects exhibited more than one strategy.

TABLE V

SOLUTION STRATEGIES IN THE POST OFFICE DOMAIN

	OO spec	Func. spec	total
good	4	2	6
monolithic	2	2	4
class. by att.	1	1	2
modular	0	0	0
func. inherit.	0	2	2
unique	0	0	0
total	7	7	14

EXPECTED SOLUTION

The expected solution for this problem consisted of an abstract class called *mail.*

This class would have subclasses *post card*, *letter*, and *package*. The class *letter* should also be an abstract class with subclasses *first class*, *second day*, and *overnight*. Each of these classes should have its own method for figuring out its postage. Additional classes outside of this hierarchy should include *post office* and *customer*.

GOOD SOLUTIONS

There were six good solutions. These solutions were characterized by having an abstract *mail* class which contained common data and functionality for letters, packages, and postcards. One of the solutions created an abstract class underneath the *mail* class called *letter*, with subclasses under that for first class, second day, and overnight letters. Two of the solutions contained a *mail* abstract class with subclasses for letters, packages, and postcards, but did not contain specific classes for first class letters, etc. Another good solution had a subclass for each type of mail directly under the abstract *mail* class. This solution contained classes such as *PostCard*, *FirstClassLetter*, *SecondClassLetter*, etc. This solution did not contain any references to packages; presumably this was just an oversight on the subject's part. Another solution combined the *post card* class with the abstract *mail* class. This is suboptimal because it complicates the abstract class; instead of just being a *mail* class, it is a *mail plus post card* class. This solution had subclasses under the *mail* class of *FirstClassLetter*, *SpecialLetter*, and *Package*. The *SpecialLetter* class contained flags for second day and overnight delivery. The last solution had classes for postcard, letter, and package, but used inheritance differently than the other subjects. Instead of having the three classes inherit from an abstract *mail* class, this solution had a *mail* class which inherited from *post card*, *letter*, and *package*. However, because this solution contained these classes and related them, even though it looked like the relationship was going the wrong way, this solution was included in the "good" category.

MONOLITHIC SOLUTIONS

These solutions were characterized by putting all types of mail into one class, usually called something like *mail*. There were four of these solutions. Three of the four solutions included functions to send and receive mail in the *mail* class. The other solution did not explicitly include these functions anywhere. Three of the four solutions included a class whose function was to compute the cost of sending mail. The other solution included a *type* field in the *mail* class which was passed to a function to determine the cost of sending a piece of mail.

CLASSIFICATION BY ATTRIBUTE

These solutions were characterized by a number of classes based on the type of mail. There were two of these solutions. Both solutions used an abstract *mail* class. Underneath this class, one solution enumerated all possible types of mail (e.g. *FirstClassLetter*, *SecondDayLetter*, etc.). The other solution combined some of the types together. Post cards were combined with the abstract *mail* class, and second day delivery and overnight delivery were combined into a *SpecialLetter* class. The only differences between all of these classes and the abstract class was that each class had its own function for computing postage, which is to be expected.

FUNCTIONAL INHERITANCE

There were two solutions which exhibited functional inheritance. These solutions were characterized by breaking the real-world relationships between objects to be able to reuse functionality or data. One solution combined the abstract *mail* class with the post card class. This complicated the idea of the abstract class. This solution also combined second day and overnight delivery into one class. The other solution also used an abstract *mail*

class. This solution had subclasses under *mail* for *SecondDay*, *LetterType*, and *Letter*. The *SecondDay* class was actually a combination of second day and overnight delivery, similar to the other solution. The *LetterType* class appeared to contain all other types of mail and seemed to be intended to calculate the cost of sending a particular type of mail. The *Letter* class appeared to contain the actual letter which would be sent. This class contained an address to send the letter to.

OTHER CLASSES

Table VI shows the number of solutions which included a *post office* class. This class was generally recognized by subjects with both types of specifications.

TABLE VI

INCLUSION OF A POST OFFICE CLASS IN POST OFFICE DOMAIN

	OO spec.		Func. spec.	
	p.o. class	no class	p.o. class	no class
good	4	0	1	1
monolithic	2	0	2	0
class. by att.	1	0	1	0
func. inherit.	0	0	2	0
total	7		7	

The following table is the number of solutions which included a *customer* class. This class models the user and how the user interacts with the system. This class was much harder to deduce from the functional specification than from the object-oriented specification.

TABLE VII

INCLUSION OF A CUSTOMER CLASS IN POST OFFICE DOMAIN

	OO spec.		Func spec.	

	customer	no customer	customer	no customer
good	4	0	0	2
monolithic	2	0	0	2
class. by att.	1	0	0	1
func. inherit.	0	0	1	1
total	7		7	

EXPERT ANALYSIS

One experienced object-oriented designer was given this problem for comparison against the novice solutions. This subject was given the object-oriented version of the specification. This design was in agreement with the expected solution. This solution included an abstract *mail* class with subclasses *post card*, *letter*, and *package*. The *mail* class contained common data and functionality, such as *send mail* and *receive mail*. The subclasses contained specific data and functionality needed for each type of mail, such as a *cost_to_send* function. This solution contained several other classes: *customer*, *post office*, *address*, and *box*. The *post office* class contained most of the functions which would be visible to a user: *purchase_stamps*, *send_mail*, and *collect_mail*. The *post office* was the driving class for this solution. The only function which was not in this class which a user would need was *cost_to_send*, which was implemented in each class individually.

VERBAL PROTOCOL

The subject began by identifying candidate classes. The subject was looking at nouns as possible classes. The subject stated that this method was a derivative of CRC. The subject identified mail, post cards, letters, packages, addresses, and weight as potential classes. Weight was also classified as a possible attribute. The subject continued by looking for verbs which would become candidate methods. The subject identified *sent*, *received*, and *weighed* as potential methods. The subject identified *first class*, *second day*, and *overnight*

delivery as possible attributes. The subject then identified more verbs: send mail, buy stamps, collect mail, find cost of sending mail. The subject also identified another noun, *post office box*. At this point, the subject hypothesized that *post office* could be a superclass for all the objects in this system. The subject continued by identifying *code* and *state* as nouns and possible classes.

At this point, the subject finished the preliminary identification and moved into a more detailed design. The subject continued by finding relationships between candidate classes and by finding base classes. The subject decided that the *mail* class was clearly going to have subclasses. The subject proceeded to name *post cards*, *letters*, and *packages* as subclasses of mail. The subject also decided that *weight* was an attribute of the *mail* class. The subject also decided that the *mail* class would have methods for *sending*, *receiving*, and *weighing* itself. The next class the subject looked at was the *letters* class. The subject decided that this class should have a *delivery class* attribute, with values of *first class*, *second day*, or *overnight*. The subject then went to the *customers* class. The subject added a *name* and a *post office box* attribute to this class. This was followed by the *post office* class, to which the subject added some methods. These methods were *cost to send*, *buy stamps*, *send mail*, and *collect mail from box*. The next class designed was the *address* class. This class contained attributes of *state*, *branch*, and *box*. After this, the subject created a generic base class of *mail* called *object*. The *object* class held global methods such as *isa* which might have been useful. The subject proceeded to design the *post office box* class. The subject mentioned that this would be a good example of a *collection* class.

The subject began looking for similarities between classes at this point. The subject also stated that this was going to be an object-based design as opposed to an inheritance-based design. The subject said that an inheritance-based design was more dependent on specific language features than an object-based design. The subject also said that an object-based design is much easier to do. The subject continued by describing the relationship of

mail and its subclasses *post cards*, *letters*, and *packages*. The subject decided that *post office* was a container class. The subject then began detailing the design of the *mail* class. The subject continued to go through the classes in more detail and finished up by actually creating the design as it would be created in a specific language. The subject made an observation that part of the fun of object-oriented programming is anthropomorphizing–thinking about inanimate objects doing things.

CHAPTER V

INSURANCE TRACKING

This problem involved two sets of subjects: novices and experts. The novice group consisted of 14 subjects. These subjects were given only a pencil-and-paper study. One expert was a subject in the verbal protocol study.

This problem asked the subjects to design an insurance tracking system. The system contained four types of transactions which needed to be modeled: "new policy", "addition to policy", "cancellation", and "surrender". These transactions types are collected into batches.

NOVICE ANALYSIS

Table VIII lists the strategies found in this problem. The same range of strategies were seen as in the train problem. Again, some subjects exhibited more than one strategy.

TABLE VIII

SOLUTION STRATEGIES IN THE INSURANCE DOMAIN

Strategy	OO spec.	Functional spec.	total
good solutions	3	0	3
monolithic	1	1	2
class. by att.	0	1	1
modular	2	2	4
func. inherit.	2	2	4
unique	1	0	1
total	9	6	15

EXPECTED SOLUTION

The expected solution combined the common data and functionality of the classes *batch* and *transaction* into an abstract class. This class contained functions such as *post* and

edit. The transaction class had subclasses underneath it for *new policy*, *addition*, and an abstract *cancellation* class. The abstract *cancellation* class combined the common features of cancellations and surrenders. It had subclasses *cancellation* and *surrender*. The expected solution also contained a *policy* class which transactions operated on and a *customer* class which enabled sales of policies.

GOOD SOLUTIONS

The good solutions were characterized by the use of an abstract class. These solutions contained abstract classes which were used to inherit common functionality and data without sacrificing consistency with the real-world problem domain. There were three of these solutions. Two of these solutions contained an abstract class (such as *abstract transaction*) which contained the common data and functionality from the *transaction* and *batch* classes. The other solution had the *transaction* class as a subclass of the *batch* class. None of the three solutions combined *cancellations* and *surrenders* with an abstract class. These two classes should have been combined because the solution stated that they were similar. The *abstract transaction* classes contained a *post* function in all three solutions. Two of the solutions also included the functions *create*, *edit*, and *print*.

MONOLITHIC SOLUTIONS

The monolithic solutions were characterized by having one class which represented both transactions and batches. These solutions combined the various types of transactions into one class which also contained functions for adding the transactions to a batch. There were two of these solutions. Both of these solutions represented transaction types as functions (e.g. *make new policy*, *add to policy*, etc.).

CLASSIFICATION BY ATTRIBUTE

This solution contained some abstraction but not as much as the good solutions. This solution abstracted the common elements from the various types of transactions to an abstract *transaction* class but did not then combine batches and transactions at a higher level. This solution created a *BasicTransaction* class which had subclasses for the various types of transactions: *NewPolicyTrans*, *AdditionToPolicyTrans*, etc. This solution had the *SurrenderedPolicyTrans* as a subclass of *CancelledPolicyTrans* which is good because the subject recognized the commonalities between these two classes. A better solution would have been to create an abstract *cancellation* class with both *cancellation* and *surrender* as subclasses of that. Also included with this solution were some notes the subject had sketched. In addition to the subclassing actually done for the *transaction* class, it appears that the subject considered doing the same type of subclassing for the *batch* class, although this was not carried out in the final solution. This solution also contained two more classes which appeared to have been derived from attributes of the specification. These classes were *BatchReport* and *TransactionReport*. Again, these classes should have been abstracted to a higher-level *Report* class and inherited functionality and data from it. Instead, both of the classes duplicated a number of functions such as *reportDate*, *fileName*, etc.

MODULAR BREAKDOWN

The distinguishing feature of these solutions is that they did not relate transactions and batches to each other but instead viewed transactions and batches as independent classes. There were four of these solutions. Three of these solutions duplicated functionality across the *transaction* and *batch* classes. The duplicated functions included *edit*, *post*, and *create*. The other solution was not clear as to what was in the *batch* class. One of the solutions represented the various types of transactions (new policy, addition, etc.) as functions within the *policy* class. Another solution represented them as subclasses of the *transaction* class. This solution also combined *cancellation* and *surrender* into one class called *Cance-*

lationOrSurrenderOfPolicy. This class held a date which the *database* class could use to determine if a policy was a cancellation or a surrender. Another solution did not appear to explicitly represent the type of a transaction any place, however, the *batch* class had a field for the type of transaction it would hold. Another class used two methods to keep track of the type of a transaction: functions within the *transaction* class and a subclass called *closedAccount* which contained surrenders and cancellations. This solution also had an *associatedWith* class which associated a policy with an annuitant and a beneficiary.

FUNCTIONAL INHERITANCE

There were four solutions which displayed functional inheritance. The solutions which displayed functional inheritance had *transaction* as a subclass of *batch* or vice versa. Thus, these solutions did not use an abstract class to unify them. Instead, they violated the principle of consistency with the problem domain to simplify the programming. These solutions shared functionality and data between the *transaction* and *batch* classes but in a non-optimal way. Using an abstract class would have increased both the depth of inheritance and the number of children measures. Two of the solutions duplicated functionality. One solution had functions such as *editTransaction* and *editBatch*, even though one class inherited the functions of the other. It should have been possible to have one *edit* function shared by both classes. The other solution which duplicated functionality was similar. The two solutions which did not duplicate functionality had all data and functions appearing in only one place in the class hierarchy. Two of the solutions used a type variable for transaction which contained the values *new policy*, *addition*, *cancellation*, and *surrender*. One of the solutions (which was also described under "good solutions") had subclasses under *transaction* for *new policy* etc.

UNIQUE SOLUTION

The unique solution used classes which reflected procedures and procedural naming conventions. This solution included classes such as *basic_functions* and *user_functions*. The functionality from the specification was stored in a class called *USER_functions*. This class contained the functions *create*, *edit*, *post*, *print_reports*, and *update*. The policy types were represented as functions within the superclass of *USER_functions*, called *Basic_transaction*. These functions were named *new_policy*, *addition_to_policy*, *policy_cancellation*, and *policy_surrender*. The other major class was *Batch_functions* which contained functions similar to those in the *USER_functions* class. This solution clearly used a functional or procedural breakdown for deciding what should be objects and classes.

OTHER CLASSES

All of the solutions except for the "unique" solution contained a *policy* class. A *policy* class is important because it gives *transactions* something concrete on which to operate. Also, policies are real-world objects. Thus, having a *policy* is another indication of good object-oriented design.

Table IX shows the number of solutions which contained a *customer* class. A *customer* class is also important because it is used to model the user. Thus, the *customer* acts as a system-level representation of the user and the functions the user can perform.

TABLE IX

INCLUSION OF A CUSTOMER CLASS IN THE INSURANCE DOMAIN

	OO spec.	Func. spec.	total
good	1	0	1
monolithic	0	1	1
class. by att.	0	0	0
modular	1	0	1
func. inherit.	1	0	1
unique	0	0	0
total	3/9	1/6	4/15

A larger number of subjects with an object-oriented specification created a *customer* class than did subjects with a functional specification. Thus, it was easier for subjects to find the *customer* class from the object-oriented specification than from the functional specification.

Table X shows the number of solutions which contained an overall driving system class such as those in the train and post office domains.

TABLE X

INCLUSION OF A DRIVING CLASS IN THE INSURANCE DOMAIN

	OO spec.	Func. spec.	total
good	1	0	1
monolithic	0	1	1
class. by att.	0	0	0
modular	1	1	2
func. inherit.	1	0	1
unique	1	0	1
total	3/9	2/6	5/15

EXPERT ANALYSIS

The subject's solution fell into the "good" category from the novice analysis. The solution contained a *policy* class which contained the data relevant to policies from the specification. The solution also contained an abstract class called *BasicOp* which included a *Posted* variable and the functions *edit*, *post*, *print*, and *update*. This class had two subclasses, *Transaction* and *Batch*. *Batch* had no further subclasses. It contained functions for accessing the list of transactions in the batch. The *Transaction* class contained data and functions for maintaining a policy number and an insurance carrier. The *Transaction* class had four subclasses, based on the type of transaction: *New Policy*, *Addition*, *Surrender*, and *Cancellation*. No data or functionality was duplicated in the class hierarchies in this solution.

VERBAL PROTOCOL

The subject began by stating that transactions and batches were important and would probably become objects. The subject then made a reference to the specification, saying that the second paragraph implied that *policy* might be an object. The subject also indicated that this paragraph implied that *transaction* might not be an object. After considering batches some more, the subject indicated that *transactions* seemed less like an object. The subject stated that the information in the "policy includes things such as...." statement from the specification indicated attributes in the *policy* class.

The subject began designing with the *policy* class. The subject decided that this class would probably not inherit anything, although that was open to change. The subject considered making both *customer name* and *address* objects but said that was not important for an initial design. The subject used domain knowledge throughout the study. At this point, the subject replaced the Social Security Number with a unique identifier because some insurance policies belong to companies instead of individuals. The subject started by making *new policy*, *addition*, *cancellation*, and *surrender* functions within the *policy* class. The subject created a *date* class after noticing the differences between surrenders and cancellations. After creating this class, the subject examined the specification again. The subject decided that the *policy* class was not going to be useful in the actual design.

The next class the subject worked on was the *batch* class. The subject began by examining the specification to find the functionality needed for batches. The subject was not sure how to proceed with this class so switched to working on the *transaction* class. At this point the subject considered making *transactions* and *batches* share a common superclass but decided to wait until after sketching both classes.

The subject switched to working on the *transaction* class. The subject decided to use subclasses under *transaction* for the various transaction types. No basis for this decision

was evident on the tape. The subject could not see instantiating the *transaction* class and so decided to make it an abstract class. The subject exhibited use of general rules, such as "these will be private instead of protected because it feels right, unless I can think of a reason to change." The subject only designed two of the subclasses at this point (*new policy* and *addition*) then said that the other two subclasses would look like the two already designed.

After finishing this part of the design, the subject switched back to working on the *batch* class. The subject decided that a batch was a conceptual array–a class that behaves like an array, though not necessarily implemented as one. The subject decided that abstracting the *create*, *edit*, and *update* functions in the transaction classes would simplify the *batch* class. Specifically, the subject commented that the *batch* class did not need to be subclassed like the *transaction* class. The subject considered making *batch* a subclass of *transaction*, but this violated the subject's real-world model. The subject had deduced (correctly) that batches could hold transactions which applied to more than one policy. If *batch* inherited from *transaction*, one of the pieces of data it would have inherited would have been a policy identifier. This was the specific reason that the subject chose not to have *batch* be a subclass of *transaction*. The next idea the subject came up with was to abstract the "posted" variable and the *edit*, *post*, *print*, and *update* functions into an abstract superclass of both the *batch* and *transaction* classes. The subject called this superclass *basic ops* (for "basic operations"). The subject quickly sketched that class. The subject then went back to the *transaction* class to make it consistent with the new superclass. The subject followed this with making the subclasses of *transaction* consistent with the *basic ops* class. Then, the subject returned to the *batch* class to make it consistent as well. After finishing the design, the subject went back to the specifications to verify that the solution contained all of the functionality that was required.

CHAPTER VI

DISCUSSION

We can account for 97% of the solutions using five strategies. These strategies are "good", "monolithic", "classification by attribute", "modular", and "functional inheritance". The other solutions fell into the "unique" category.

GOOD SOLUTIONS

The good solutions were characterized by abstraction. All of the good solutions contained at least one abstract class. Some of the good solutions contained more than one abstract class. Good solutions did not duplicate data or functionality throughout the class hierarchies.

MONOLITHIC SOLUTIONS

The monolithic solutions were characterized by a single class which covered multiple real-world objects. These solutions did not display object-oriented characteristics such as inheritance or encapsulation.

CLASSIFICATION BY ATTRIBUTE

These solutions were characterized by an overuse of classes. These solutions multiplied classes unnecessarily. An example from the train system was subclassing *ticket* into *first class* and *first class with reservation*. This type of solution is suboptimal because to modify an attribute ("reservation", in this case) requires the program to delete the existing subclass (*first class* or *first class with reservation*) and replace it with the other subclass. Thus, the program has to know and distinguish between the two subclasses which violates the principle of encapsulation.

MODULAR BREAKDOWN

These solutions were characterized by independent, unrelated classes. The common parts of some of the classes could have been combined into an abstract class, as was done in the good solutions. This would have improved the "depth of inheritance tree" and "number of children" metrics. This would also have resulted in less duplication of code.

FUNCTIONAL INHERITANCE

These solutions were characterized by an inheritance relationship which violated the real-world object relationships. This is a suboptimal design.

UNIQUE SOLUTIONS

These solutions were characterized by purely procedural thinking. These solutions contained classes which would have been more logically created as methods. Examples of this included the *Basic_operations* and *exchangeTicket* classes.

CONCLUSION

The type of specification, object-oriented or functional, appeared to have an influence on the type of the solution. Table XI contains the data broken down by specification type. The table does not include data for the train problem because it only included one type of specification.

TABLE XI

SOLUTION STRATEGY BY SPECIFICATION TYPE

	object-oriented spec.	func. spec.	total
good	7	2	9
monolithic	3	3	6
class by att.	1	2	3
modular	2	2	4
func. inherit	2	4	6
unique	1	0	1
total	16	13	29

The largest difference is evident in the "good" solutions. There are 7 good solutions (58%) with an object-oriented specification versus 2 (15%) with a functional specification. The only other large difference is in the "functional inheritance" category–there are 2 (17%) with an object-oriented specification versus 4 (31%) with a functional specification. This indicates that a specification which explicitly stresses the objects in the problem domain can improve the chances of finding a good object-oriented solution.

DOMAIN EFFECTS

Table XII categorizes the different types of solution based on problem domain across all novice solutions.

TABLE XII

SOLUTION STRATEGY BY PROBLEM DOMAIN

	insurance	train	post office
good	3	4	6
monolithic	2	9	4
class by att.	1	9	2
modular	4	7	0
func. inherit	4	11	2
unique	1	1	0
total	15	41	14

The expected results based on familiarity with the problem domain would be that insurance would have the fewest good solutions, the train system would be in the middle, and the post office would have the most good solutions. The intuition about the post office having the most good solutions was supported by the data; 43% of the solutions in the post office domain were good. However, the insurance and train domains were the opposite of what was expected. This may have been due to the small number of subjects in the insurance domain where small variations can be magnified, relative to the train domain. The post office domain was unique in not displaying all types of strategies. The "modular" and "unique" strategies were not found in this problem domain. A logical conclusion from this is that the familiarity with the problem domain allowed the programmers to find natural, real-world relationships between the objects.

The effects of domain knowledge have been recognized in the object-oriented programming community. This has resulted in the job of "domain analyst" being created. A domain analyst is a person who is expert in a particular area, such as insurance systems or post offices. This person has been trained to assist programmers in understanding the problem domain.

SPECIFICATION EFFECTS

Eight of the subjects were given two problems: these problems varied both in problem domain (insurance or post office) and in specification type (functional or object-oriented). Each subject was given both a functional and an object-oriented problem specification, one in each problem domain. Of these eight subjects, one used the same strategy for both types of specifications. All of the other subjects used a different strategy between the two problems. There was no consistency across specification type or problem domain. The subjects were influenced by both the type of specification and the problem domain. Six of the subjects had a "good" solution to one of the problems. 5 of these subjects were using an object-oriented

specification to create these solutions, and 1 subject used a functional specification. 5 of the good solutions were in the post office domain, and 1 was in the insurance domain. 4 of the 6 good solutions were created from an object-oriented specification of the post office domain.

DESIGN DECISIONS IN VERBAL PROTOCOLS

The experts all specifically considered whether or not to use an abstract class in their solutions. Two of the four experts exhibited serendipitous design behavior–they moved up and down the class hierarchy as opposed to designing top-down or bottom-up. Only one expert designed the hierarchy top-down. The experts were also concerned with whether something should be an attribute rather than an object. All the experts checked their design when they had finished it to make sure that it met the specification. These are decisions which novices need to learn to consider.

FUTURE RESEARCH

The next step in this research is to create a tool for several sample problem domains which could critique designs in the given domains. The data from this study could be used to help build such a tool. The tool should classify the solution and the strategies used and help the designer overcome a procedural background.

Another area of research is reusability. This study could be modified to give the subjects a second problem. After finishing their initial design, subjects could be asked to modify the design to include an additional problem-domain constraint or object. An example of this would be adding a "daily pass" to the train domain.

Another area of study would be to investigate use of a large, standard class library to see if this has an impact on the designs people produce. This could also be combined with testing to see if implementation has an effect on the object-oriented quality of a design.

SUMMARY

I identified a finite number of strategies used by procedural programmers in an object-oriented environment. The five strategies identified were "good", "monolithic", "modular", "classification by attribute", and "functional inheritance". These strategies were successful in categorizing solutions across three problem domains and two specification types. I found that familiar problem domains and object-oriented specifications contributed to producing good object-oriented designs.

REFERENCES

1 Manns, Mary Lynn & Carlson, David A. (1992). Retraining Procedural Programmers: A Case Against Unlearning. presented at OOPSLA 1992.

2 Scholtz, Jean & Wiedenbeck, Susan. (1990). Learning Second and Subsequent Programming Languages: A Problem of Transfer. *International Journal of Human-Computer Interaction*, *2*(1), 51-72.

3 Dvorak, Joseph L. & Moher, Thomas G. (1991). A Feasibility Study of Early Class Hierarchy Construction in Object-Oriented Development. In J. Koenemann-Bellivene, T. Moher, & S.Robertson (Ed.), *Empirical Studies of Programmers: fourth workshop*, Norwood, N.J.: Ablex, 23-35.

4 Green, T.R.G., Gilmore, D.J., Blumenthal, B.B., Davies, S., & Winder, R. (1992). Towards a Cognitive Browser for OOPS. *International Journal of Human-Computer Interaction*, *4*(1), 1-34.

5 Coad, Peter & Yourdon, Edward (1990). *Object-Oriented Analysis*. Englewood Cliffs, N.J.: Yourdon press.

6 Beck, Kent & Cunningham, Ward (1989). A Laboratory for Teaching Object-Oriented Thinking. *OOPSLA '89 Proceedings*, 1-6.

7 McGregor, John D. & Korson, Tim (1993). Supporting Dimensions of Classification in Object-Oriented Design. *Journal of Object-Oriented Programming*, *5*(9), 25-30.

8 Rosson, Mary Beth & Alpert, Sherman R. (1990). The Cognitive Consequences of Object-Oriented Design. *Human-Computer Interaction*, *5*, 345-379.

9 Lieberherr, Karl J. & Holland, Ian M. (1989). Assuring Good Style for Object-Oriented Programs. *IEEE Software*, *6*(5), 38-49.

10 Chidamber, Shyam R. & Kenerer, Chris F. (1993). A Metrics Suite for Object-Oriented Design. submitted for publication.

11 Ward, Paul T. (1989). How to Integrate Object-Orientation with Structured Analysis and Design. *IEEE Software*, *6*(2), 74-82.

12 Wu, Quanfeng (1991). *Knowledge Transfer among Programming Languages*. Unpublished doctoral dissertation, Carnegie Mellon University, Pittsburgh, PA.

13 Katz, Irvin R. (1991). Assessing Transfer of a Complex Skill. *Proceedings of the 14th Conference of Cognitive Science Society*.

14 Détienne, Françoise (1992). Acquiring Experience in Object-Oriented Programming: Effects on Design Strategies. NATO Advanced Workshop.

15 Wu, Quanfeng, and Anderson, John R., (1991) Knowledge Transfer among Programming Languages, *Proceedings of the 14th Conference of Cognitive Science Society*.

16 Soloway, E., Ehrlich, K., Bonar, J., & Greenspan, J. (1982). What Do Novices Know About Programming? In A. Badre and B. Shneidermann (Ed.), *Directions in Human/ Computer Interaction*, Norwood, N.J.: Ablex, 27-54.

17 Rist, Robert S. (1986). Plans in Programming: Definition, Demonstration, and Development. in *Empirical Studies of Programmers*, Ablex, Norwood, N.J., 28-45.

18 Rist, Robert S. (submitted for publication). Formal and Informal Design Strategies in Programming. *International Journal of Man-Machine Studies.*

19 Pennington, Nancy (1992). Comparing Procedural and Object-oriented Design. poster at CHI '92.

20 Singley, Mark K., Carroll, John M., & Alpert, Sherman R. (1991). Psychological Design Rationale for an Intelligent Tutoring System for Smalltalk. In J. Koenemann-Bellivene, T. Moher, & S. Robertson (Ed.), *Empirical Studies of Programmers: fourth workshop*, Norwood, N.J.: Ablex, 23-35.

21 Fischer, G., Girgensohn, A., Nakakoji, K. & Redmiles, D. (1992). Supporting Software Designers with Integrated, Domain-Oriented Design Environments. *IEEE Transactions on Software Engineering.*

APPENDIX A

PROBLEM SPECIFICATIONS

SPECIFICATION FOR THE TRAIN PROBLEM

This is a simulation of a train system.

In this system a customer can buy tickets, cancel reservations, change their reservations, and exchange their ticket for one to a different destination.

Tickets come in three forms: first class with a reservation, first class without a reservation, and second class. The differences between first and second class are price and quality. A first class ticket with a reservation guarantees the customer an assigned seat and also costs extra. Reservations are made for a specific time, in the customer's name. Tickets have a specific starting point and destination. Tickets can have a reservation added after they have been bought.

There are also monthly passes which can be purchased for first or second class. The system keeps a record of people buying monthly passes. Monthly passes don't include reservations. A customer with a monthly pass can make a reservation for an additional fee.

Prices:

Tickets	one-way	round trip
1st class, reserved	$15	$25
1st class	$10	$15
2nd class	$5	$7

Passes, monthly	
1st class	$200
2nd class	$100
reservations:	$3/per reservation

FUNCTIONAL SPECIFICATION FOR THE INSURANCE PROBLEM

Insurance Tracking

The basic operations which need to be included in the system are: create transactions and batches, edit transactions and batches, post transactions and batches, print reports of posted and unposted transactions and batches, update transactions (correcting mistakes in a transaction after it has been posted–will require posting another transaction to fix it), and add transactions to batches.

A transaction can be one of four types: new policy, addition to policy, cancellation of a policy, or surrender of a policy (a cancellation which occurs more than 2 years after the sale date).

A batch is a homogeneous group of transactions.

Policy information includes things such as customer name, address, social security number, policy amount, identifier, and policy carrier (i.e. who sells it).

OBJECT-ORIENTED SPECIFICATION FOR THE INSURANCE PROBLEM

Insurance Tracking

A policy contains a policy identifier (a tag to uniquely identify each policy) and a policy coverage amount. A policy needs to be associated with an annuitant (the person who is covered by the policy) and a beneficiary (the person who is paid if the annuitant dies).

There are four basic transaction types which need to be included in the system. These types are: new policies, additions to policies, cancellations of policies, and surrenders of policies.

A new policy needs customer information (such as name, address, and social security number), policy amount, policy identifier, and which kind of policy it is (i.e. who the carrier is).

An addition to a policy needs some way of identifying the policy to add to and the amount to add to the policy.

A cancellation needs a way to identify the policy. A cancellation results when the customer ends the policy coverage within 2 years after the sale date.

A surrender is like a cancellation, but it occurs more than 2 years after the sale date.

The user needs to be able to do certain operations on all types of transactions. These operations include create, edit, update (correcting mistakes in a transaction after it has been posted–will require posting another transaction to fix it), add transactions to batches, post transactions, and print reports of posted and unposted transactions.

A batch is a homogeneous group of transactions. The operations needed on batches are create, edit, post, and print reports of posted and unposted batches.

FUNCTIONAL SPECIFICATION FOR THE POST OFFICE PROBLEM

This is a simulation of a post office.

Customers need to be able to find the cost of sending mail, buy stamps, send mail, and collect any mail in their post office box.

The post office delivers mail according to its address. An address is a name and a 9-digit code. The first two digits of the code are the state, the next three are the post office branch and the last four are the post office box.

Letters can be delivered second day or overnight.

Postal rates:	
postcards	$0.19
letters, first class	
first oz.	$0.29
each additional oz.	$0.15
second day	$3.00
overnight	$10.00
packages	$1.00 / pound

OBJECT-ORIENTED SPECIFICATION FOR THE POST OFFICE PROBLEM

This is a simulation of a post office.

Mail comes in three different forms: postcards, letters, and packages. All types of mail have to and from addresses. All types of mail also have a weight. Mail can be sent, received, and weighed. An address is a name and a 9-digit code.

Letters come in three types: first class, second day, and overnight. The differences are how long it takes the letter to be delivered.

Customers have a name and a post office box. They can find the cost of sending mail, buy stamps, send mail, and collect any mail in their post office box.

The post office contains post office boxes. It can deliver mail according to the address. The first two digits of the code are the state, the next three are the post office branch and the last four are the post office box.

Postal rates:	
postcards	$0.19
letters, first class	
first oz.	$0.29
each additional oz.	$0.15
second day	$3.00
overnight	$10.00

APPENDIX B

GLOSSARY

children: a *class* can be set up to contain the same data and functionality as another class–a class that does this is called a *child* of another class. The child *inherits* data or functionality from it's *parent* class.

class: a collection of data and functionality, or state and behavior.

inherit: to include data or functionality from one *class* in another.

inheritance tree: the collection of inheritance relationships existing in a program.

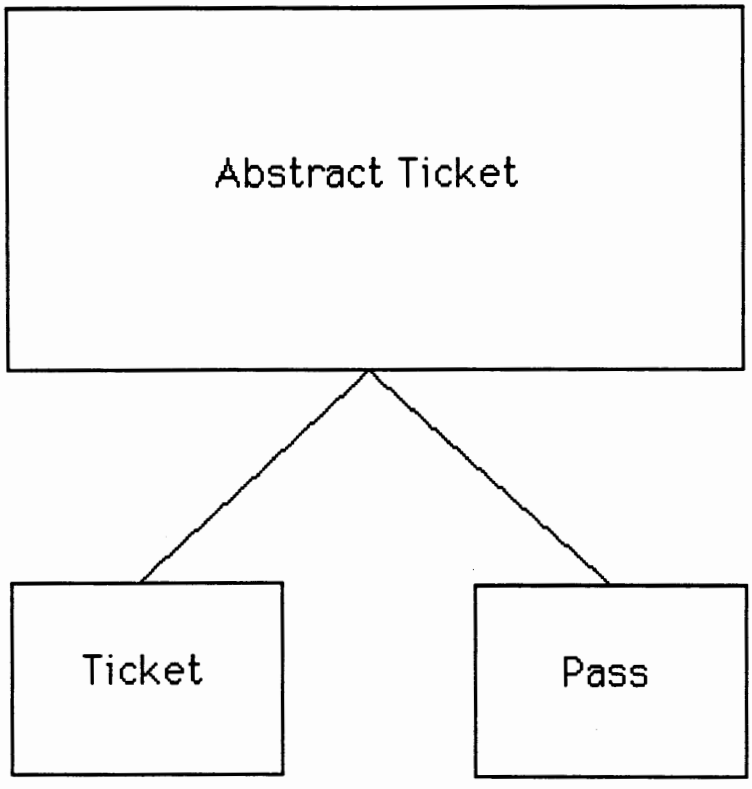
parent: a *class* which provides data or functionality which another class *inherits*.

Smalltalk class hierarchy: this is a standard model of an *inheritance tree*. It involves having a single *class* at the top with all other classes as *children* of that class.

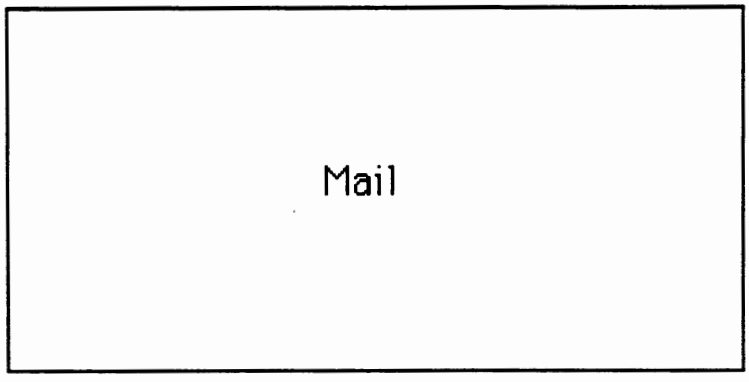
APPENDIX C

EXAMPLE SOLUTIONS

GOOD / EXPECTED SOLUTION, TRAIN SYSTEM



MONOLITHIC SOLUTION, POST OFFICE



FUNCTIONAL INHERITANCE, INSURANCE