Fundamental Patterns for Logic Design

Robert F. Zant
School of Information Technology
Illinois State University
Normal, IL 61790, USA

Abstract

Students new to information technology are often at a loss as to how to transform a problem statement into a program design. A number of different approaches have been proposed to provide students more guidance than is typically found in introductory texts. A new approach is presented that is based on two fundamental patterns in computing—the Input-Process-Output pattern and the Initialization-Loop-Termination pattern. An example application of the approach is presented.

Keywords: IS 2002.5, program design, novice programmers, teaching programming, HIPO

1. INTRODUCTION

Students typically perceive the beginning programming course as a language course--that is, they characterize the course as a "VB," "COBOL," "C++," or "JAVA" course. To them, learning the syntax and semantics of the language is a primary focus (Barr, Holden, Phillips 1999). Instructors, on the other hand, view the course much more broadly with the language being just a tool used to support the other, more important topics in the course--topics like problem analysis, program design, and OO concepts.

Perhaps the reason students focus on the language is precisely because it does underpin the other topics and it is more concrete and easier for the students to learn. However, in designing curriculum, instructors focus on the non-language topics because they are seen as the most important. This relationship between language and concepts has bred an active discussion of just how topics should be organized in the introductory course. One reason for this difficulty is that the topics comprise a non-linear set. The more one knows about each topic, the easier it is to understand the others. This is often viewed as comprising a threshold level of knowledge that must be obtained before "the pieces fall into place." Once this threshold has been reached, students suddenly "catch on" and comprehension replaces confusion. This phenomenon has been widely recognized as the problem of closing “the gap between a problem statement and a programmed solution” (Lane and VanLehn 2003), as the student not knowing where to begin (Proulx 2000; Maris, VanLengen, Lucy 2000; Adams and Frens 2003), and as giving the student atoms and letting them “figure out how to build molecules” (Rabb, Rasala, Proulx 2000).

In recognition of this interdependency of topics, introductory courses should be designed to repeat concepts to allow the accumulation of knowledge to take place. This is often accomplished by covering the concepts in a spiral approach where the concepts are repeated, but in more and more depth. The repetition of concepts affords students the opportunity to gain the threshold of knowledge needed but, since the complexity level is increasing, the student must be able to make progress or fall hopelessly behind. Ad hoc information indicates drop rates from 25% to 50% are not uncommon in the introductory course. Bouvier recently reported a 40% attrition rate at the University of Houston-Downtown (Bouvier 2003).

The dialog concerning the teaching of introductory programming has been enlivened in recent years by the advent of object oriented (OO) concepts and languages being taught in the course. Some of the points that have been raised are:

- Entering students already have experience using sophisticated software (Wester, Sint, Kluit 1997).
- Past experiences lead students to expect to work with GUI and graphics (Rabb, Rasala, Proulx 2000).
- Exercises should be intellectually stimulating and even exciting (Hadjerrouit 1998; Rasala 2000).
- Students are most comfortable working with concrete ideas and rules that are to be followed (Zant 2001).
- The details associated with learning and using a language and development environment can be overwhelming (Proulx 2000; Wester, Sint, Kluit 1997).
- Students find logic structures and OO concepts difficult to master (Stix and Mosley 2002).
- Debugging is an "enigma wrapped in a puzzle" (Lang 2002; Hristova, Misra, Rutter, Mercuri 2003).
- Even students who grasp the details have difficulty with problem solving. They are frustrated by the sense of "not knowing where to begin" (Proulx 2000).

The remainder of this paper will focus on this last point. Other proposals for dealing with this issue will be reviewed and a new technique using Action Tables and IPO/ILT Charts will be introduced and its use demonstrated through an example.

2. OTHER PROPOSALS

A number of proposals have been advanced to assist students in designing programs. The proposals seek to provide a "starting point" and a procedure that can be used by a novice student in creating a program design given a problem statement. One, high-level, approach is to have students follow a template for the structure of classes that comprise a program. The Model/View approach (Kluit, Sint, Wester 1998; Christensen and Caspersen 2002; Bruce, Danyluk, Murtagh 2001) decomposes classes into two categories. The first type, Model classes, provides the applications functionality. The second type, View classes, provides the user interface. The importance of this approach goes beyond providing a standard structure for students to use. In initial assignments, the interface classes can be provided to students so that the students are then only responsible for programming the logic for the required functionality. This approach spares the student from becoming involved with the minutia required for implementing user interfaces in OO languages.

A related approach is suggested by Koffman and Wolz (Koffman and Wolz 1999). They structure programs into an Application class and one or more Support classes. The Application class contains the “static main” method that instantiates a support class and then contains minimal logic to invoke the required methods in the Support class. This approach is combined with the use of an IO Toolkit (Wolz and Koffman 1999) to simplify, for the student, the implementation of the user interface.

Lane and VanLehn propose an approach reminiscent of the Guild approach of the master and the apprentice (Lane and VanLehn 2003). Their approach, *Coached Program Planning* (CPP), pairs a tutor and a student to “collaborate to build a natural-language-style pseudo-code solution” for a problem statement. The CPP dialogue repetitively follows a four-step pattern until pseudo-code has been developed for all functional requirements in the problem statement.

1. identify the next programming goal
2. describe a way for attaining the goal
3. select pseudo-code steps to attain the goal
4. sequence the pseudo-code steps appropriately

This approach provides students a guide experienced in problem solving until they have gained enough experience to individually design programs.

Adams and Frens suggest the "Where do I begin?" problem be solved with a procedure they call *object centered design for Java* (Adams and Frens 2003). This is a four-step process where the student begins by rewriting the problem statement using key words such as *program, keyboard*, and *screen*. Step two is to identify the *nouns* (objects) in the problem statement followed by identifying the *verbs* (operations). The final step is to apply the operations to the objects to construct a "static main" method. I/O classes are provided to simplify the programming of the user interface.

The Noun/Verb paradigm was extended by Reichgelt and Kung and embedded in a process to identify classes, attributes, and behaviors (Reichgelt and Kung 2002). They provide a very well-developed process for analyzing nouns to identify and refine classes. Students then identify attributes and the behaviors associated with them. This methodology gives students an easy beginning point, that of listing nouns, and provides detailed steps for refining the "nouns" into classes.

Bergin integrates the Noun/Verb approach with design patterns in a nine-step methodology for designing the class structure for a problem (Bergin 1998). Proulx presents a number of design patterns that are used "to help students focus on mastering reasoning and design skills" (Proulx 2000). Maris uses one of those patterns, the input-process-output pattern, as the basis for a design tool, a Summary Table, for students to use to collect, classify, and compare the components of a problem statement (Maris, VanLengen, Lucy 2000). Students place components of a problem into a table with six columns. The first column is used to list each major task that must be accomplished in the problem. For each task, the remaining columns contain the input data, its source, the output data, its source, and the trigger event for the task. Students may begin with any column in the table so that the approach allows the freedom to begin with the components that the student most easily recognizes. The Summary Table is used as a starting point for the student to analyze and gain an understanding of the problem. Once sufficient knowledge of the problem has been obtained, the student switches to other design techniques such as UML.

3. THE PROPOSED APPROACH

The approach presented here, using Action Tables and IPO/ILT Charts, integrates two fundamental patterns in program design. Unlike techniques such as activity diagrams that are strictly procedural in nature, Action Tables and IPO/ILT Charts combine structural and procedural views of a problem. This is easier for a student to use since the student does not have to identify the precise sequence of actions at the same time as the actions themselves are identified. In other words, the student does not have to initially think procedurally.

The *Input-Process-Output* (IPO) pattern is, perhaps, the first pattern to be used in computing (Gustavson and Choolfaian 2000). The IPO pattern can be applied to a program as a whole or to a subsection of code. It is the basis for analysis and design techniques such as System Flowcharts, HIPO Charts, and Use Case Diagrams.

Another fundamental pattern of computing is the *Initialization-Loop-Termination* (ILT) pattern. This pattern can also be applied to an entire program or to a task. The pattern recognizes that a task typically consists of some actions that are taken initially, followed by actions that are carried out repetitively based on some condition, and then completed by actions that follow the loop. The ILT pattern is endemic to such analysis and design techniques as Program Flowcharts and Activity Diagrams.

4. ACTION TABLES

An Action Table is a technique that is used to bridge between analysis and design. It is used to classify actions along an analysis dimension and a design dimension. The student must only identify and classify required actions in the problem statement.

An Action Table contains three columns. The first column contains a list of actions to be performed in the system. The second column classifies each action as input, process, or output (analysis dimension). The third column classifies each action as initialization, loop, or termination (design dimension).

The Action column is filled in first. The student reviews the problem statement to identify the system processing requirements and rephrases them as individual actions that must be performed. An "I," "P," or "O" is entered in the second column for each action depending on whether it is an input, processing, or output action. Likewise an "I," "L," or "T" is entered in the third column for each action depending on when the action must be performed.

If the student is undecided as to how to classify an action, it is reexamined to determine if it can be subdivided into two or more actions. For example, "get interactive response" may originally be identified as a single action. But, upon reexamination, it may be expressed as two actions, "display prompt" and "get value." The newly identified actions can then be classified as an output action for displaying a prompt and as an input action for getting the value.

After classifying all identified actions, the table is reviewed along with the system's requirements to assure that all relevant actions have been identified and correctly classified.

5. IPO / ILT CHARTS

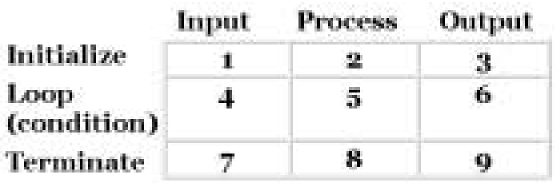
An IPO/ILT Chart is a tool used in the design of a system to classify actions required in the system and to partially specify the sequence required for the execution of the actions. Actions are classified in a two-way classification scheme that organizes them generally according to their logical sequence of execution.

The chart contains three columns for classifying each action as an Input, a Process, or an Output action. Often, input actions logically precede related process actions that, in turn, logically precede related output actions. But, this is not always the case, e.g., when a prompt message is displayed on a screen (output) and then the related response is entered (input) and, finally, processed (process).

The chart contains three rows for classifying actions: actions performed to initialize a logic sequence, actions performed within a loop, or actions performed at the termination of a logic sequence. This classification (ILT) is in strict accordance with the logical precedence for related actions. That is, all the *initialize* actions will be executed before any of the *loop* actions. And, all of the *loop* actions will be completed before the *terminate* actions are executed.

The two dimensions of the IPO/ILT Chart produce nine different categories to aid in the analysis of actions. To use the chart, enter each action on the Action Table into the appropriate cell in the IPO/ILT Chart. Actions within a cell are entered in the sequence in which they are to be executed relative to the other actions in the same cell. In some cases, the sequence is critical. In others, the order may not matter. The order required must be determined given the statement of the problem. The precise sequencing of the actions from different cells into a coherent sequence will be accomplished after the IPO/ILT Chart is completed.

Structure of the IPO / ILT Chart



	Input	Process	Output
Initialize	1	2	3
Loop (condition)	4	5	6
Terminate	7	8	9

After completing the IPO/ILT Chart, the entries are reviewed with the Action Table to confirm that all actions from the table have been entered correctly. Also, the condition governing the execution of the loop must be specified. Both the minimum number of times the loop will be executed (zero or one) and the condition under which the loop will be terminated must be determined. It is not necessary for all nine cells to contain actions. One or more cells may be empty.

6. EXAMPLE

The following is an example for calculating the present value of a stream of annual returns. A system requirements statement for the problem is given along with a corresponding Action Table and IPO/ILT Chart. Some pedagogical implications for the use of this approach are also given.

System Requirements

Design the logic to calculate the present value for an investment. Input for the calculation will be the amount of the initial investment (C0), the number of years the investment will be active (n), the net cash flow for each year (Ci), and the discount factor (d). The present value is calculated as:

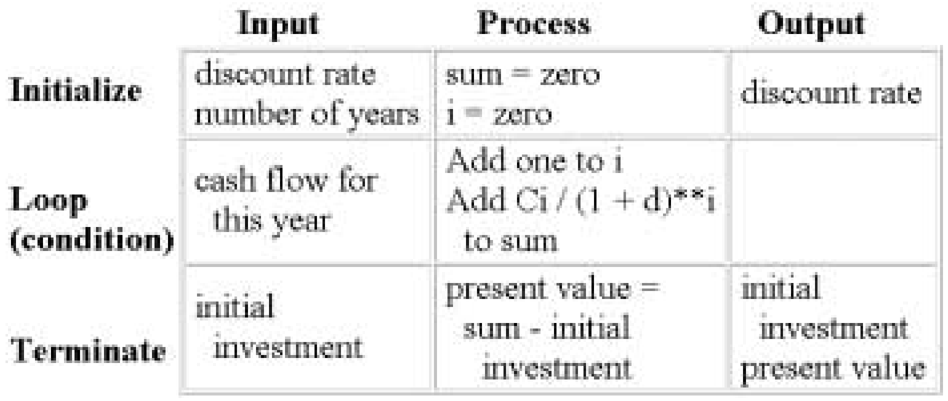
PV = sum over n [Ci / (1 + d) ** i] - C0

Output will consist of the value of the initial investment, the discount rate, and the computed present value of the investment.

Action Table

ACTION	IPO	ILT
Get discount rate	I	I
Get number of years	I	I
Get cash flow for year	I	L
Get initial investment	I	T
Display discount rate	O	I
Display initial investment	O	T
Display present value	O	T
sum = zero	P	I
i = zero	P	I
Add 1 to i	P	L
Add Ci / (1+d)**i to sum	P	L
Present value = sum - initial investment	P	T

IPO / ILT Chart



	Input	Process	Output

Initialize	discount rate
	number of years
		sum = zero
		i = zero
			discount rate

Loop (condition)	cash flow for this year	Add one to i Add Ci / (1 + d)**i to sum	
Terminate	initial investment	present value = sum - initial investment	initial investment present value

The use of the IPO/ILT Chart provides a focus for discussion of several design decisions. For example, actions within a category may have to be executed in a particular sequence or the sequence of execution may not matter. In the *Initialize/Process* cell either of the two actions could be done first. But, given that the exponent in the calculation (i) is initialized to zero rather than to one, the two actions in the *Loop/Process* cell must be done in the specified order.

The chart also demonstrates that in some cases there is a choice of categories for an action. That is, the program would function well with alternative placements of an action. In this example, the input action of reading the "initial investment" could be done with other input actions in the *Initialize/Input* cell rather than where it is in the *Terminate/Input* cell. Choices should be made based on some logical criteria, such as producing clear and easily maintained code. In this case the choice was made by placing actions as late as possible in the logic sequence, i.e., a just-in-time rule. This will keep actions that have a sequence dependency as close together as possible in the logic and ultimately in the software code.

Finally, the IPO/ILT Chart offers a convenient vehicle for discussing the difference between logical models in the problem space and in the solution space. In the example, the initialization of the variables "sum" and "i" are solution space actions and would not typically appear in the problem statement and hence would not appear in an IPO/ILT Chart in "problem space".

Once the student is satisfied that the IPO/ILT Chart correctly reflects the logic of the problem, the next step is to design a coherent sequence of logic that explicitly sequences the actions contained in the IPO/ILT Chart. This detail logic is designed one row at a time from the IPO/ILT Chart; first the *Initialize* row, then the *Loop* row, and finally the *Terminate* row.

Logic within each row is generally developed starting with actions entered in the *Input* cell, then the *Process* cell, and finally the *Output* cell. However, actions within a row may not always follow this strict logical sequence. In particular, all actions within the *Process* cell will not necessarily logically precede all actions within the *Output* cell, etc. The actions within a row must be analyzed carefully to determine their logical sequence.

The resulting sequence of actions can be expressed in pseudo-code or directly into code. In developing code from the IPO/ILT Chart additional statements will have to be added to the logic depending on the language used--for example, the declaration of variables.

7. CONCLUSION

Beginning information systems and computer science students often find that they do not know where to begin in analyzing a problem statement. The techniques presented in this paper, Action Tables and IPO/ILT Charts, provide a methodology for novice computing students to use in transforming a problem statement into a program design and ultimately into a computer program. Traditional techniques such as flowcharts, data flow diagrams, and activity charts are strictly procedural in nature and have not proved easy for students to use. The methodology presented requires that students first classify actions. The cross-classification as both IPO and ILT provides guidance in the sequencing the actions. The IPO/ILT Chart provides a vehicle for discussing design decisions and for understanding the role of logical models.

8. REFERENCES

Adams, Joel and Jeremy Frens, 2003, "Object Centered Design for Java: Teaching OOD in CS-1." ACM SIGCSE'03, February 19-23, pp. 273-277.

Barr, Matthew, Sam Holden, Dave Phillips, and Tony Greening, 1999, "An Exploration of Novice Programming Errors in an Object-Oriented Environment." SIGCSE Bulletin, Vol. 31, No. 4, pp. 42-46.

Bergin, Joseph, 1998, "Patterns of Object-Oriented Design for Novices." HTTP://csis.pace.edu/~bergin/patterns/ design.html.

Bishop, Judith, 1997, "A Philosophy of Teaching Java." ACM Joint Conference of IT in CS," pp. 146.

Bouvier, Dennis J., 2003, "Pilot Study: Living Flowcharts in an Introduction to Programming Course." ACM SIGCSE'03, February 19-23, pp. 293-295.

Bruce, Kim B., Andrea Danyluk, and Thomas Murtagh, 2001, "Event-driven Programming is Simple Enough for CS 1." ACM ITiCSE'01, pp. 1-4.

Christensen, Henrik Baerbak and Michael E. Caspersen, 2002, "Frameworks in CS 1 – a Different Way of Introducing Event-driven Programming." ACM ITiCSE'02, June 24-26, pp. 75-79.

Gustavson, Fran and Stephen Choolfaian, 2000 "On a New Teaching Paradigm for Information Systems." ISECON 2000.

Hadjerrouit, Said, 1998, "A Constructivist Framework for Integrating the Java Paradigm into the Undergraduate Curriculum." ACM ITiCSE'98, pp. 105-107.

Hristova, Maria, Ananya Misra, Megan Rutter, and Rebecca Mercuri, "Identifying and Correcting Java Programming Errors for Introductory Computer Science Students." 2003, ACM SIGCSE'03, February 19-23, pp. 153-156.

Kluit, Peter G., Marleen Sint, and Frank Wester, 1998, "Visual Programming with Java: Evaluation of an Introductory Programming Course." ACM ITiCSE'98, pp. 143-147.

Koffman, Elliot and Ursula Wolz, 1999, "CS1 Using Java Language Features Gently." ACM ITiCSE'99, pp. 40-43.

Lane, H. Chad and Kurt VanLehn, 2003, "Coached Program Planning: Dialogue-Based Support for Novice Program Design." ACM SIGCSE'03, February 19-23, pp. 148-152.

Lang, Bob, 2002, "Teaching New Programmers: A Java Tool Set as a Student Teaching Aid." International Conference on the Principles and Practice of Programming in Java, pp. 95-100.

Maris, Jo-Mae, Craig VanLengen, and Rick Lucy, 2000, "A Design Tool for Novice Programmers." ISECON 2000.

Proulx, Viera K., 2000, "Programming Patterns and Design Patterns in the Introductory Computer Science Course." ACM SIGCSE 2000, pp. 80-84.

Rabb, Jeff, Richard Rasala, and Viera K. Proulx, 2000, "Pedagogical Power Tools for Teaching Java." ACM ITiCSE 2000, pp. 156-159.

Rasala, Richard, 2000, "Toolkits in First Year Computer Science: A Pedagogical Imperative." ACM SIGCSE 2000, pp. 185-191.

Reichgelt, Han and Hsiang-Jui Kung, 2002, "A Methodology for Teaching Object Oriented Design and a Preliminary Evaluation." Proceedings of the 2002 Conference for Information Technology Curriculum.

Stix, Allen and Pauline Mosley, 2002, "Cognitive Complexities Confronting Software Developers Utilizing Object Technology." ISECON 2002.

Wester, Frank, Marleen Sint, and Peter Kluit, 1997, "Visual Programming with Java: an Alternative Approach to Introductory Programming." ACM ITiCSE'97, pp. 57-58.

Wolz, Ursula, 1997, "Language Considerations in a Goal-Centered Approach to CS I and II: Java, C, or What?" Journal of Computing in Small Colleges, Vol. 12, No. 5, pp. 12-20..

Wolz, Ursula and Elliot Koffman, 1999, "simpleIO: A Java Package for Novice Interactive and Graphics Programming." ACM ITiCSE'99, pp. 139-142.

Zant, Robert F., 2001, "Problem Analysis and Program Design Using Subsystems and Strategies." ISECON 2001.