Problem Analysis and Program Design: Using Subsystems and Strategies

Robert F. Zant[1]
Department of Applied Computer Science
Illinois State University
Normal, IL, 61790, USA

Abstract

Although there has been a substantial amount of research on methods for developing computer programs, students new to the art of programming continue to find it difficult to transform a problem statement into a functional program. This paper reviews the difference between the novice's and the expert's approach to programming, and presents two techniques--the IPO Diagram and Composition Strategies that novices can use to gain a better understanding of problem analysis and its impact on program design.

Keywords: Analysis, design, programming, IPO diagrams, composition strategies

Although there has been a substantial amount of research on methods for developing computer programs, students new to the art of programming continue to find it difficult to transform a problem statement into a functional program. The time-honored techniques such as HIPO Charts, Flowcharts and Hierarchy Charts do not seem to provide sufficient guidelines to lead the student from analysis to design. Several alternative approaches have been used with reported success. Hohmann, et al. (1992) used a methodology based on "goals and plans" with a high school Pascal class. They reported that students could complete more assignments within a term using the technique and that the students' programming abilities were transferable to large-scale projects. Another approach that stresses the use of patterns is becoming popular and is being used in some texts (Epp 2001). Syntax-less approaches, such as Karel the Robot (Pattis 1995) and Iconic programming (Calloni 1997) have also been shown to be effective.

1. REVIEW OF RESEARCH

Past research has shown that expert programmers not only master the syntax and semantics of a computer language but also learn patterns for solutions to commonly encountered problems. In addition to these technical skills, experts also gain application domain knowledge that they bring to bear on specific systems. Dreyfus (Dreyfus 1982) found that experts view a problem in the context of its domain and draw on their experience in similar situations to formulate a solution. Reitman (1965) noted that the situation-oriented point of view led experts to spend more time analyzing a problem than did a novice. The novice tends to approach a problem not in the "gestalt," but rather from a context-free perspective. Since the novice does not have a "tool-kit" of patterns derived from experience, the novice relies on a rule-based approach, e.g., use an IF statement when there is a choice of two actions, that tends to focus on individual details.

Bloom's (Bloom 1956) taxonomy of the cognitive domain can provide insight into the difference in the way a novice approaches problem solving and the approach of an expert. Bloom defines six levels of learning that must be accomplished in succession. Novice programmers would begin at the ***Knowledge/Comprehension*** levels in learning computer concepts, syntax, and semantics. They then often encounter difficulty in moving to the next level, ***Application***. At this level novices must use their factual

[1] rfzant@ilstu.edu

knowledge in the construction of programs in the context of familiar situations, e.g., creating a program very similar to an example in the text. The fourth level, ***Analysis***, is also problematic for students. At this level the student attains a level of understanding required to debug code, i.e., analyze a run-time error, and to begin developing solutions in less familiar situations. The fifth level, ***Synthesis***, is well beyond the novice's abilities. At this level the programmer can develop solutions to new and complex problems. The sixth level, ***Evaluation***, is reserved for the expert. At this level the expert can develop new algorithms and can select from alternative platforms, algorithms, etc., based on their suitability to the problem.

A conclusion that can be drawn from Bloom's taxonomy is that it would be difficult, at best, for a novice to become an expert without progressing through each of the six levels. In fact, Shackelford and Badre (1993) found that students in an introductory Pascal class were more successful at applying patterns based on ***constructive*** rules that focused on language constructs (level 3) than in applying ***descriptive*** rules that focused more on the context of the problem (level 4 and 5). Linn (Linn 1992) also found that new programmers had difficulty learning and using patterns. Roberts (Roberts 2001) has observed that students have difficulty learning JAVA, in part, because of the extensive libraries, i.e., canned patterns, that must be learned. Novice programmers thus do best in working with language constructs in a rule-based environment and less well in learning context-based patterns. This is as we would expect based on Bloom's taxonomy.

This paper proposes that a sequence of techniques be used in learning to analyze a problem and then to design a program. The techniques to be used in the analysis phase are: IPO Diagram, Composition Strategies, and HIPO Charts. The techniques recommended for use in design are: Hierarchy Chart and either Pseudopodia or Flowcharts. Only the IPO Diagram and Composition Strategies will be presented since the other techniques are well known. These two techniques provide rule-based models for students to use in the analysis of a problem. Students gain experience in analyzing salient characteristics of systems and see the importance of understanding characteristics of the application system that influence the ultimate design of the program.

2. IPO DIAGRAMS

IPO Diagrams are conceptually very simple. They simply embody the input-process-output metamodel that is well known and is the basis for the HIPO technique. IPO Diagrams are used to decompose a system. It differs from other techniques used in analysis in that it seeks to decompose a system into subsystems rather than into functions.

The process begins by thinking of the system in its entirety. The model would be a single input-process-output sequence:

I → P → O

The decomposition rule is to then ask if the transform "P" is a simple transform of input to output. Jackson's (Jackson 1975) concept of data structure correspondence is useful here. The first question to be asked is: "For the defined set of data (keyboard input, file, or database) does 'P' process each item of input sequentially to produce the output stream?" If this is "no," such as when only a subset of the input is to be processed, then the system is divided into two subsystems. There are two cases: when some preprocessing must be completely accomplished before the primary process can be carried out, e.g., sorting--

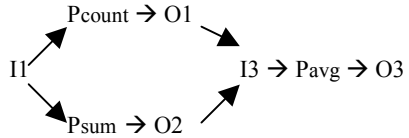
I1 → P1 → O1\ I2 → P2 → O2

And when the output from the first subsystem can be directly piped to the second, e.g., record selection--

I1 → P1 → O1/ I2 → P2 → O2

The second question to be asked is: "Does the format of the output stream depend only on the input data?" This would be the case when, for example, input from the keyboard is displayed directly on the screen or when data is entered and only a total is displayed. The answer would be "no" in cases such as formatted multi-screen displays or when there are multiple output streams. When the answer is "no," the system must be further subdivided as done above.

The third question deals with the process "P" itself to determine if it is a simple or compound process. "Could 'P' be subdivided into two or more processes that could individually process the input to produce a partial outcome for the program?" For example, a process to read a stream of data and calculate the average value could be decomposed into a process to count the number of data points and another process to calculate the sum and then a third to calculate the average value, as--



Once the problem has been decomposed into simple subsystems, the network of subsystems must then be partitioned into implementation units. The above system could be partitioned into one, two, or three implementation units. Considerations for this partitioning are: geographical constraints, platform

constraints, timing constraints, and personnel/complexity constraints.

3. COMPOSITION STRATEGIES

Next, the composition strategy for each of the partitions must be determined. There are four basic composition strategies for combining subsystems: Disjoint, Aggregate, Embed, and Integrate. [Note: the use of the term ***composition strategy*** is a little different than as used by Spohrer and Soloway (Spohrer 1986)].

The **Disjoint** strategy is to separately develop subsystems that are logically connected by the timing of their execution. For example, on Windows systems the Scandisk program should always be executed before the Defrag program. The IPO Diagram would be:

I1 → Pscan → O1\I2 → Pdefrag → O2

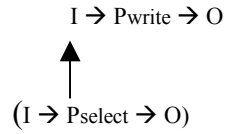
These two subsystems are implemented in Windows as two separate programs because Scandisk is useful without Defrag, although Defrag should not be run without first running Scandisk. Consequently, it is up to the user to run Scandisk before defragmenting a disk.

The **Aggregate** strategy automates the execution of dependent programs by the use of a control module or script. For example, the following VB script will run the Scandisk program and then the Defrag program.

```
Set WshShell = WScript.CreateObject
("WScript.Shell")
ReturnVal = WshShell.Run
("C:\WINDOWS\SCANDSKW.EXE", True)

Set WshShell = WScript.CreateObject
("WScript.Shell")
ReturnVal = WshShell.Run
("C:\WINDOWS\DEFRAG.EXE", True)
```

The **Embed** strategy is called "program inversion" in the Jackson methodology. It is a strategy to encapsulate a subsystem. The interface between the subsystem and the main system is implemented by the passing of a status value. Take the example of a program that is to read a file, select certain records, and display the selected records. The selection subsystem can be designed as an embedded system that reads the file and returns only selected records to the main system. Then, the main system can be designed as if all records were to be displayed since it will only receive selected records from the embedded module. The IPO Diagram would be:



A sample JAVA program for this strategy is:

```
import java.io.*;
public class select2 {
     static String item;
     static int price;
     static DataInputStream df = null;

     public static void main(String[] args) throws IOException {
          char status = selection ('i');
          PrintWriter pw = new PrintWriter (new FileWriter("report.txt"),
               true);
          while (item != null) {
               pw.println (item + ", " + price);
               status = selection (status);
          } // End While
          selection ('c');
          pw.close();
     } // End main

     public static char selection(char status) throws IOException
          switch (status) {
          case 'i':
               df = new DataInputStream(new FileInputStream("ss.data"));
               status = 'o';
               // no break here, read first item and price
          case 'o':
               while (((item = df.readLine()) != null) &&
                    ((price = df.readInt()) < 100)) {
               } // End while
               break;
          case 'c':
               df.close();
```

```
            } // End switch
        return status;
        } // End selection

} // End Class
```

The fourth strategy, **Integrate**, is the most commonly used and also the most problematic strategy. Soloway (Soloway 1986) found that novice programmers have the most difficulty developing programs using this strategy (what he called "merged plans"). This strategy seeks to combine the logic constructs from multiple subsystems into one system. An IPO Diagram for the above program would be:

I → Pselect & Pwrite → O

Example JAVA code is:

```
import java.io.*;
public class select1 {

    public static void main(String[] args) throws IOException {
        String item;
        int price;
        DataInputStream df = new DataInputStream(new FileInputStream
            ("ss.data"));
        PrintWriter pw = new PrintWriter (new FileWriter("report.txt"),
            true);
        while ((item = df.readLine()) != null) {
            price = df.readInt();
            if (price >= 100) {
            pw.println (item + ", " + price);
            } //End If
        } //End While
        df.close();
        pw.close();
    }//End main

}//End Class
```

This strategy produces the most compact code. But, when the code is not arranged and documented properly, this compactness creates code that is difficult to debug and maintain. The problem is caused by code from different subsystems, i.e., different purposes, being mixed together with redundant code being discarded. To make changes in the code, the programmer must be able to distinguish the impact of the proposed changes on each of the integrated subsystems.

4. CONCLUSION

Two techniques, IPO Diagrams and Composition Strategies, have been presented that are used to analyze programming problems. The techniques correspond to the levels of Bloom's taxonomy that are appropriate for novice programmers. Furthermore they can be applied to relatively simple problems that are typically assigned in a first course on programming. The techniques complement more traditional techniques such as HIPO Charts and Hierarchy Charts. They are applied before these other techniques so that they bridge the gap between the problem statement and the analysis of detail logic.

5. REFERENCES

Bloom, B. S., et al., *Taxonomy of Educational Objectives: Handbook I: Cognitive Domain*, Longmans, Green and Company, 1956.

Calloni, B. A., and Bagert, D. J., "Iconic Programming Proves Effective for Teaching the First Year Programming Sequence," *Proceedings of SIGCSE Technical Symposium on Computer Science Education*, 1997.

Dreyfus, S. E., "Formal Models versus Human Situational Understanding: Inherent Limitations on the Modeling of Business Expertise", *Office: Technology and People*, August 1982.

Epp, Ed C., *Prelude to Patterns in Computer Science Using JAVA*, Franklin, Beddle & Associates, 2001.

Hohmann, L., et al., "SODA: A Computer-Aided Design Environment for the Doing and Learning of Software Design," *Computer Assisted Learning: Proceedings of the Fourth International Conference on Computers and Learning*,

Springer-Verlag Lecture Notes in Computer Science 602, 1992.

Jackson, M. A., *Principles of Program Design*, Academic Press, 1975.

Linn, M. C., "How can Hypermedia Tools Help Teach Programming?" *Learning and Instruction*, volume 2, 1992.

Pattis, R. E., *Karel the Robot: A Gentle Introduction to the Art of Programming*, John Wiley & Sons, 1995.

Reitman, W. R., *Cognition and Thought*, John Wiley & Sons, 1965.

Roberts, Eric, "An Overview of MiniJava", *Proceedings of SIGCSE Technical Symposium on Computer Science Education*, 2001.

Shackelford, R. L. and Badre, A. N., "Why Can't Smart Students Solve Simple Programming Problems?" *International Journal of Man-Machine Studies*, June 1993.

Soloway, Elliot, "Learning to Program = Learning to Construct Mechanisms and Explanations," *Communications of the ACM*, September 1986.

Spohrer, James C., and Soloway, Elliot, "Novice Mistakes: Are the Folk Wisdoms Correct?" *Communications of the ACM*, July 1986.