A Design Tool for Novice Programmers

Jo-Mae B. Maris
College of Business Administration, Northern Arizona University
Flagstaff, Arizona 86011-5066
jo-mae.maris@nau.edu

Craig A. VanLengen
College of Business Administration, Northern Arizona University
Flagstaff, Arizona 86011-5066

and

Rick Lucy
College of Business Administration, Northern Arizona University
Flagstaff, Arizona 86011-5066

ABSTRACT

Most program design methods are intended for experienced programmers. Beginner friendly program design methods date back to procedural languages, such as Pascal and Basic. These methods lack connections to objects and events since the languages contained neither objects nor events. This paper presents a summary table and a sketch to get novice programmers started in the process of designing a program. The table organizes information about the program requirements and aides in creating a design for a program that may contain events and objects. The sketch represents the calling relationships among the modules in the program. The table and the sketch can be use with an existing method, such as pseudocode.

The tools enhance existing methods of design. A new method is not proposed. The most important philosophies in developing the tools were simplicity and guidance. The table guides the student's design efforts and is simple. The columns collect data about what the program does, when it does its tasks, and what data it uses. The rows relate tasks, events, and objects. The table prompts identification of objects and events and makes high-level functionality stand out. The high-level functional design captured by the table is made explicit in the relations sketch.

Keywords: Program design, design tool, novice programmers, teaching programming

1. PROBLEM

Novice programming students frequently ask, "Where do I start?" How many times has the question been asked after the teacher has presented structured-design, object-oriented design, or the universal modeling language? Does the problem lie with the teacher or with the methods?

At least one method did not specify a starting place in its initial presentation. The method was stepwise design:

> In each step, one or several instructions of the given program are decomposed into more detailed instructions. This successive decomposition or refinement of specifications terminates when all instructions are expressed in terms of an underlying computer or programming language, … (Wirth, 1971).

Wirth did provide more guidance in his Pascal User Manual and Report: "In the early stages, attention is best concentrated on the global problems, and the first draft of the solution may pay little attention to the details" (Jensen, 1974).

Most of the methods give starting points:

- "The first step in actual class design is to find the primary objects" (Arnow, 2000).
- "… make a model that defines the key domain classes in the system" (Erikkson, 1998).
- "Identify the classes and objects at a given level of abstraction" (Booch, 1991).
- Investigate the problem domain: observe first-hand; listen actively; check previous OOA results; check other systems; read, read, read; and prototype (Coad, 1991).

- Rules for developing a proof or program:
  1. Do the single option available in the simple case of only one option;
  2. Choose the complex option;
  3. Start on the most complicated side.

  That is start with the hard job first (Dijkstra, 1988).
- The first business of design is therefore to translate the specifications into the fixed formats of a set of working documents (Data Flow Diagrams, Data Dictionary, Transform Descriptions, and Data Structures Charts…) (DeMarco, 1979).

The student programmer may be able to recite the definition of terms used in the preceding guidelines, but to **make use** of the concepts may be beyond the level of learning programming for the student. A programmer needs experience to grasp the hard job, level of abstraction, problem domain, and domain classes. For the novice, all of the jobs are hard. Level of abstraction, problem domain, and domain classes are terms the novice has memorized. Two sets of authors discussing object-oriented design recognized this problem. Rumbaugh, Blaha, Premerlani, Eddy, and Lorensen acknowledge the problem directly:

> "The content of an object model is a matter of judgment …" (Rumbaugh, 1991).

Judging is in the problem-solving level of the cognitive domain of learning (Daniell, 1990). "The knowledge level forms the base upon which the application level is built, and the application level forms the base for the problem-solving level" (Daniell, 1990). Hence, beginning students, who are still memorizing terms, are in the knowledge level that includes "define, recite, repeat, and restate" among its activities (Daniell, 1990).

Coad and Yourdon recognized the problem indirectly by pointing out that through experience objects become readily apparent:

> As analysts experienced in applying OOA across widely divergent problem domains, we recognize certain patterns across systems. And so at times it might seem that the Class-&-Objects are 'just there for the picking' (Coad, 1991).

However, student programmers would not have the experience of Coad and Yourdon.

Another recognition of the need for experience to design programs using existing methods comes from DeMarco in the chapter titled "Transition into the Design Phase":

> "When you're done with this chapter you won't know how to do a Structured Design, unless you knew that already" (DeMarco, 1979).

Yet, more evidence of the need for experience in the use of some program-design methods comes from Erikkson and Penker:

> … there is no "right" solution for all circumstances. Of course, some solutions will prove better than others, but only experience and hard work will result in that knowledge (Erikkson, 1998).

We believe the problem is with the methods. Dijkstra articulates a problem that this and no other method can overcome: "Not all teachable topics are learned by all students (enrollees)" (Dijkstra, 1988).

1. PROPOSAL

We believe that a novice programmer needs a simplified approach to program design. We propose using a find, list, and order approach. The approach gives the student a place to start that he understands. The starting place may be different for each student. This approach does not propose a new method. Rather the proposal is to provide a tool that will guide identification of data needed by existing methods and to classify and organize that data so it becomes information the student can use with existing methods. The student should switch to an existing method when he or she comprehends the information the existing method uses.

2. DESIGN APPROACH

The driving force behind our approach is the old dictum, "Keep It Simple, Stupid," also know as Ockham's razor.[1] Given the complexity of developing programs for an event-driven environment using an object-oriented language, keeping it simple is essential. The simplicity of the approach also answers the novice programmer's question, "Where do I start?" Collecting data and listing data does not require experience. Sorting data requires classifications and comparison. Classification is an application level skill. Comparison is a problem-solving level skill. Using the progression from knowledge level skills to application level and then to problem-solving level enables, the student to progress in understanding so the data manipulated will become information.

This design approach has few strictly defined components or rules. The components are objects, events, tasks, and data. The rule is work with the summary table and relationship sketches until you have identified the information needed to use an existing method, such as pseudocode to specify the low-level

[1] "They followed the emperor to Munich (Germany) in 1330, where Ockham wrote fervently against the papacy in a series of treatises on papal power and civil sovereignty. The medieval rule of parsimony, or principle of economy, frequently used by Ockham came to be known as Ockham's razor. The rule, which said that plurality should not be assumed without necessity (or, in modern English, keep it simple, stupid), was used to eliminate many pseudo-explanatory entities." (Bechett, Dave. http://wotug.ukc.ac.uk/parallel/www/occam/occam-bio.html University of Kent at Canterbury, UK, 1994.)

functionality. The simplicity in the approach is an intentional choice to minimize a student's feeling of inadequacy and ignorance.

An object is an integrated package or bundle of properties and behaviors. Objects respond to events. Properties describe the characteristics, qualities, appearance, values, or data built-in an object. Behaviors refer to the actions, methods, processes, operations, or code built-in an object. The programmer can generate an object in a visual development environment or declare an object. The object created has the properties and methods intrinsic to that object available without declaring them or coding them separate from the object.

Events are interactions between the user and objects, objects and objects, or code and objects. The objects are usually in the user's interface, but the objects may be system objects, such as printers. Events send messages to the controlling module. The controlling module uses the event's message (or signal) to determine which default behaviors and event procedure to execute. The behaviors and event procedures are blocks of code that define tasks.

Tasks are work that needs to be done. The tasks may be performed by objects' methods or by user-defined code. When a task is an identifiable block of code, the task may be called a module, such as methods, event procedures, or user-defined procedures. In this sense, a module is a coherent block of code, not just a convenient container for holding coherent blocks of code, as are the *modules* in Visual Basic. In our approach, the term module has more the meaning of the modules in a hierarchy diagram or a method of an object. When a task is small, it may be a program statement or a portion of the code in a module. Whether a task is large or small, eventually its performance must be prescribed as ordered program statements or code.

A module can be a child (or called module) or it can be a parent module. There are two types of parent modules: a controlling module and a calling module. A controlling module could be referred to as a controller, kernel, core unit, primary logic, main module, or any similar term that conveys a software entity that is in control of when subordinate modules are called. Frequently, the controlling module is predefined, such as the controlling logic built into Visual Basic, the script interpretation features of an Internet browser, or services of an Internet server that interprets ASP pages or invokes CGI programs. However, the controlling module could be user-defined, such as the primary logic module in a procedural language program. A calling module launches or invokes another module. A calling module may be a controlling module or a user-defined module.

Data are the numbers, details, specifics, or representations of facts manipulated in the performance of the work.

Now that the terms have been defined, we present our approach in the stages of the system development life cycle. Hoffer, George, and Valacich list the stages of the SDLC, as "Project Identification and Selection, Project Initiation and Planning, Analysis, Logical Design, Physical Design, Implementation, and Maintenance" (Hoffer, 1998). Our approach concentrates on selected portions of the SDLC:

- Problem definition from the Project Identification and Selection stage,
- Initial program and GUI design from the Logical Design stage,
- Program and GUI design refinement from the Physical Design stage,
- Program and GUI construction from the Implementation stage, and
- Testing and debugging from the Implementation stage.

The approach we propose is for learning to design programs and their interactions with their interfaces. It is not intended to cover the entire SDLC. The programs students develop tend to be small and rarely if ever used or evaluated in the context of a changing business environment, so Project Initiation and Planning, Analysis[2], and Maintenance would not be relevant to the programming projects of novice programmers.

Problem Definition

In our approach, the problem definition is primarily a scaled down combination of Project Identification and Selection, Project Initiation and Planning, Analysis stages performed by the teacher. The teacher provides the requirements for the program. A simplified requirements document is the central tool for this stage. The emphasis is on providing the students with the information necessary for developing a program. Using simplified requirements documents for assignments allows the students to experience working from requirements documents. The document used in our approach is called a formal problem definition as described in class notes during the early 1980's from Computer Science Department, University of Wisconsin—La Crosse.

A formal problem definition consists of five parts: Overview, Input Expected, Output Required, Normal Example, and Unusual and Error Conditions. More information on the Formal Problem Definition is available upon request.

Initial Design

The purpose of the initial program design is to give the student programmer a place to start. During this stage of

[2] "Analysis" in the sense of developing the system requirements is not relevant to the design of novices' programs. In our approach, the instructor would supply the system requirements, so the students would begin working in the Logical Design stage.

the design, the student studies the formal problem definition to recognize and record the tasks (high-level functionality) and objects described in the formal problem definition. The tasks and objects are recorded in a Summary Table. This stage could begin with listing the output required, and then identifying the data necessary to produce the output. The Summary Table includes columns for listing the major tasks the program is to perform, the data manipulated by the tasks, and the major objects. One layout of the Summary Table is shown in Figure 1.

Figure 1: Summary Table

Program's Major Tasks	Input Data	Get	Output Data	Put	Trigger Obj/Event
{Cryptic description of task}	Item A	GUI	Item A	GUI & File	New record command button /click
	Item B	Ask user	Item C	Print	
{Another cryptic description}	Item D	GUI or File			Form/load
{Last cryptic description}			Item D	File	Called by Form/unload and new record command button/click

Summary Table: The summary table is composed of three areas: tasks, data, and triggers. The tasks' column is labeled "Program's Major Tasks" in Figure 1. The tasks column gives students a place to list the tasks, behaviors, or operations described in the formal problem definition. The label for this and any column in the table could be changed, if a different label would be more useful to the student using the table. For example, if a student finds the label "Model's Behaviors" more descriptive than "Program's Major Tasks," then the student should be encouraged to use the revised label.

The triggers area of the summary table is labeled "Triggers, Obj/Event" in Figure 1. The triggers' column provides an area for accumulating the object and event combinations that trigger behaviors or event procedures that perform the tasks listed in the "Program's Major Tasks" column. That is the triggers represent the messages or signals used by the controlling module to select what behaviors and event procedure to execute. If a student finds identifying objects easier than identifying tasks or events, this column could be split into two columns: (1) Objects and (2) Events. The new objects' column could be the first column filled. The student would then need to identify the behaviors (tasks) and events associated with the objects.

The data area contains four columns in Figure 1. The columns are "Input Data," "Get," "Output Data," and "Put." The "Input Data" column provides a location for noting the data required to perform a task. The "Get" column gives the source of the data used to perform a task, "Input Data." The "Output Data" column provides a location for noting the data produced by a task. The "Put" column specifies destination of the data produced by a task, "Output Data." The labels "Get" and "Put" were chosen because the labels are short. If a student prefers the labels "Source" and "Destination," then use "Source" and "Destination." Some students may find the data area the easiest to complete first since the formal problem definition contains sections for input and output. Again, the student should start with the easiest column.

Process Used with Summary Table: The process described in this section is not rigidly structured. At any time during this process, the student may switch to an existing design method. The purpose of the tool is to provide the student a means for collecting data and converting the data to information that is useful in using existing methods. Thus, once the student has obtained sufficient information to use an existing design method, the student should switch to developing the program's design with that method's tools.

The first step in completing the table is to fill-in one column. The column would be the one the student finds the easiest to identify its contents in the formal problem definition. For example, the students might find identifying tasks the easiest column to complete. Figure 2 shows an example of a student starting with the "Program's Major Tasks" column.

Once the one column is completed, then the students would identify the content of another column. The data entered in the second column should relate to the data in the completed column by row. For example, the student might identify the output of each task. The data about the output would also be available in the formal problem definition. For example, the user may want to be able to either enter a client's address or have it retrieved from a file.

The trigger column may be the most difficult for the students to complete because it may not be included in

the formal problem definition. The possible objects in some interfaces are limited, such as loading of a Web page for some JavaScripts, or extensive, such as the wide array of controls and events that can trigger event procedures in Visual Basic.

After completing the table, the user's interface can be designed. Once the table is completed, the student has identified the major objects to appear in the program and on the user's interface. Then the user's interface design becomes a matter of arranging the objects in a productive and attractive manner.

Design Refinement

Once the user's interface is designed, the student is ready to concentrate on the details of performing the tasks. During the design refinement stage, the student moves from modeling the system as an abstraction to modeling the details of the system. The tools used to develop this detail model may be pseudocode and a graphical representation of the modules similar to a hierarchy chart, but modules may occur in the chart more than once and at different levels in the tree. Figure 4 shows an example of a relation sketch that resembles a hierarchy chart, but is not a hierarchy chart. Recall this graphic is a sketch to help a student visualize the calling relations among the modules or behaviors. Some students may want to color a module that repeats, so the repeated module becomes more obvious. Granted, repeating modules in a hierarchy chart is not allowed. However, this is not a hierarchy chart.

The sketch is to help the student. If a graphic similar to an object diagram, a systems diagram, or a data flow diagram does a better job of enlightening the student, then use it. However, do not let rules of a tool hinder the student's understanding. At this point the student is still trying to understand the relationships, so those relationships can be represented in a more rigorous manner. It is important not to impose rigor before the student understands the relationship.

The activities in this stage begin with identifying each module and its relationships with other modules. This procedure was begun in the initial design stage by identifying the tasks and their triggers. Now the tasks and the triggers need names. The names will become module, method, or subroutine names. The names can be written on the Major Tasks' Information form.

To clarify each module's relationship with other modules, a chart or map of the modules can be drawn. For a Web-based program, the drawing might resemble a site map that included code modules. For a single-form Visual Basic program, the drawing would resemble a hierarchy chart. However, the graphic should emphasize the relationships among task modules and controlling modules. Therefore, the graphic should include representations of controlling modules, event procedures, and user-defined code. If the relationships are most easily represented and understood by including a module more than once and at different levels of the tree, then do it. See the "Example" section for an illustration of one possible graphic representation of the relationships among modules. The precise appearance and presentation is not the important part of this step. The importance of this step is to solidify the relationships among the task modules and controlling modules.

After establishing the relationships among modules, each module should be refined using pseudocode and stepwise refinement. As new subroutines are identified, they should be added to the graphic. The utility subroutines could be put in the relationships sketch a number of times or be represented as modules at the bottom of the sketch with lines from calling modules leading to the utilities. These utility constructs are sometimes called octopuses because the lines leading to a utility resemble tentacles reaching into the orderly structure of a hierarchy chart. Once all of the modules are fully specified in pseudocode, the student is ready to proceed to the next stage, program construction.

Program Construction

Constructing the program refers to creating the user's interface and writing the code. The primary concern of this method in this stage is the translation of design into language-specific and presentation-specific constructs. For example, in Visual Basic a scrollable output area would be constructed using a text box control tool and setting the text box properties so that Multiline is True, Scrollbars equals 2, and Locked is True. Another example of the conversion would be to select the correct instruction syntax to construct a loop planned in the pseudocode while keeping the code structured.

Testing and Debugging

This modeling approach does not have specific recommendations for testing and debugging.

3. CONCLUDING REMARKS

Most program design methods are intended for experienced programmers rather than beginners. The summary table and graphic relationships tools presented give the instructor additional means to help novice programmers collect and organize the data used in existing design methods. Since the tools do not assume programming experience, the instructor can assign students knowledge level tasks before requiring application or problem-solving level tasks. By making, the tasks commensurate with the students' level in learning frustration in learning programming should be reduced. Thus, we posit that the tools presented herein improve instruction of programming by facilitating students' learning processes.

4. REFERENCES

Arnow, David M. and Gerald Weiss, 2000, Introduction to Programming Using ***java***™: An Object-Oriented Approach. Menlo Park, Addison-Wesley, p.142.

Booch, Grady, 1991, Object Oriented Design with Applications, Fort Collins, Benjamin/Cummings Publishing Company, Inc., p.190.

Coad, Peter and Edward Yourdon, 1991, Object-Oriented Analysis,2nd ed. Englewood Cliffs, YOURDON Press, p.58.

Daniell, Elizabeth O., 1990, TIPS: Teaching Improvement Project Systems for Health Care Educators. Lexington, Kentucky, Center for Learning Resources, College of Health Professions, University of Kentucky, p.34 & p.46.

DeMarco, Tom 1979, Structured Analysis and System Specifications. Englewood Cliffs, Prentice-Hall, p.24 & p.297.

Dijkstra, Edsger W., 1988, "Formal Derivation of Programs," Notes from a workshop in Monroe, LA, May 16-20, p.1 & p.9.

Erikkson, Hans-Erik and Magnus Penker, 1998, "Design Java Apps with UML," JavaPro, June/July, pp.1-2.

Hoffer, Jeffrey A., Joey F. George, and Joseph S. Valacich, 1998, Modern Systems Analysis and Design. Menlo Park, Addison-Wesley, p.25.

Jensen, Kathleen and Niklaus Wirth, 1974, Pascal, 2nd ed. New York: Springer-Verlag, p.52.

Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen, 1991, Object-Oriented Modeling and Design. Englewood Cliffs, Prentice Hall, p.47.

Wirth, Niklaus, 1971, "Program Development by Stepwise Refinement," Communications of the ACM, Vol. 14, No. 4, pp.221-227.