A model of programming

Lindsey Ford
Department of Computer Science
University of Exeter
Exeter EX4 4PT
UK

(lindsey@uk.ac.exeter.dcs)

(Extended Abstract submitted to PPIG-7 Annual Workshop)

Visual Programming has been shown to have useful potential and it is now worth considering how it might be used to provide a coherent programming environment [Hils (1992), Price et al (1993), Ford (1993)]. Although the reviews of prototype systems clarify the class of underlying programming language that is being supported (for example, procedural, object-oriented, or functional) they do not clarify what underlying model of programming is being supported and it is thus difficult to determine to what extent the VP systems reviewed will help overcome the long-recognised problems of program comprehension. For example, many systems reviewed by Hils (1992) and Price et al (1993) deal primarily with small programs that can be developed simply: the comprehension problems associated with scale of problem/solution are not addressed nor are the methods by which large programs are arrived at discussed. This latter point is disconcerting since it seems that VP systems are in danger of merely being a substitute for a text editor and a textual programming language. It leaves an uncomfortable feeling that the visual programming joy ride will provide research psychologists with many, presumably happy, hours analysing why programmers have difficulty understanding a multiplicity of views and images. Meanwhile the professional software development community will bemoan that another research venture has failed to deliver the promised productivity and reliability gains.

While it may well be possible to replace a programming language such as C++ by a visual one it does not provide any guarantees with respect to program comprehension. Indeed any VP environment that does not heed analysis and design steps simply cannot provide information to programmers that years of research has deemed are relevant to program comprehension. To what extent these steps can effectively be accommodated in a traditional CASE or VP environment is open to question. The important thing, however, is to ask the question.

Objected-oriented programming (OOP) is an obvious candidate for the VP treatment - mapping objects to graphics is seductively easy - and a number of research efforts (no names, no pack drill ... but mine included) are going in this direction. What model of programming do they support? Unfortunately OOP, unlike procedural programming which has a relatively long history and is bound to the notion of an algorithm, has yet to reveal a clear and unambiguous method [Booch (1993), Mills (1993)] and has some weaknesses in that it appears users find it less easy to ``have a good representation of data flow and control flow'' [Detienne (1990)]. On this basis the VP treatment appears doomed.

Below is briefly presented a model of how programming in OOP can be undertaken with tenets of program comprehension in mind. Three tenets I gleaned many years ago, and nothing in 30 years experience of programming has persuaded me otherwise, is that to understand an imperative program you need to be able to relate three models: a task model (the program's teleology), a procedural model (its flow of control), a data model (its data structure and flow) [Lukey (1981)]. The desired understanding is used to further develop or enhance the program, debug it, or crucially to use it as the basis for some other development effort. The latter use frequently requires someone other than the original author to make the effort.

For systems that result in more than one program the following method would be buttressed with various additional activities. The method below is primarily aimed at single (but perhaps large-scale) program solutions to problems that have been thoroughly analysed. The method is best viewed as three stages - analysis, design, and implementation.

The analysis stage involves three activities:

- conceptual analysis,
- determine system inputs and outputs,
- task analysis.

Conceptual analysis yields a list of concepts, each one of which turns out to be either a class, an attribute of a class, or an irrelevance to classes and their attributes. Task analysis provides a sequenced list of task descriptions, each of which is decomposed to lower level descriptions. The system inputs and outputs are a high-level description of data flow and are an expression of what the program's purpose is - namely to transform the inputs to the outputs. Lower-level details are later required to show how the program's internal data flow achieves it. The task descriptions, and inputs and outputs, provide the first inkling of the relationships of program purpose to data flow and program purpose to control flow.

The design stage has these steps:

- identify classes and their relationships, - identify attributes of classes,
- construct a class model,
- construct a task-data table,
- construct a task flow of control model.

The first three steps are concerned with constructing the static part of the data model and as yet are not concerned with the procedural model. The task-data table relates each task description to a system input/output (where possible) and to any class attributes. In the table are noted any class attributes that are changed or used. The basis of the data flow at a class level is established. The task flow of control model is no more than flow of control constructs (choice and loop) mapped onto the task analysis to give a procedural model.

The implementation stage involves:

- defining the class model,
- implementing procedural logic.

Implementing the class model involves defining classes and their relationships and developing appropriate member functions to meet the requirements of the task-data table. The latter and the task flow of control model are used together to implement the procedural logic, which will invariably include defining global functions, global and local variables - the data flow model is thus accomplished. The importance of these steps is that the data and procedural models are developed hand in hand and in full view of the task model.

In summary, a programming model for OOP has been presented that takes account of central tenets of program comprehension. It should thus serve as a starting point for the design of any OOP environment. Of particular interest to the VP enthusiasts is that there's an awful lot of text associated with task models. It is the ad hoc comment annotations found in programs that provide glimpses of this task model which is so crucial to most programmers for comprehension purposes. This is not to say that such models cannot be represented by animations or visual representations enhanced with text. But the model will be needed.

References

Booch G (1993) *Object-oriented analysis and design* (2nd ed), Benjamin/Cummings Inc, California.

Detienne F (1990) "Difficulties in designing with an object-oriented language: an empirical study", in D Diaper et al (eds), *Human-Computer Interaction - INTERACT'90*, 971--976, IFIP.

Ford L (1993) "How programmers visualize programs", Research Report 271, Department of Computer Science, University of Exeter, Exeter, UK.

Hils D D (1992) "Visual languages and computing survey: data flow visual programming languages", *Journal of Visual Languages and Computing*, 3(1):69--101.

Lukey F J (1981) "Comprehending and debugging computer programs", in M J Coombs and J L Alty (eds), *Computing skills and the user interface*, pages 201--219, Academic Press.

Mills Z (1993) "Object-oriented analysis and design methods: a comparison" Technical Report CSTR-93-2, Department of Computer Science, University of Brunel, Uxbridge, Middlesex, UK.

Price B A, Baecker R M, and Small I S (1993) "A principled taxonomy of software visualization", *Journal of Visual Languages and Computing*, 4(3):211-266.