# What is the appropriate abstraction for understanding and reengineering a software system?

Arun Lakhotia

The Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, LA 70504
(318) 482-6766, -5791 (Fax)
arun@cacs.usl.edu

***Appeared in the "Reverse Engineering Newsletter," IEEE Computer Society, Number 7, Sep 1994, pp. 1-2.***

To understand a software system for maintaining it a model — requirement, design, or some other view — is created by abstracting certain details away from the source code. The model plays a central role in any automated support for software maintenance. The choice of abstraction — the notation or technique used to represent a model — therefore becomes a crucial decision in the development of any maintenance tool.

In building a research agenda on tools to support software maintenance, I like other researchers in the same boat, have often wondered what the most appropriate abstraction of a software system would be. Having weighed the pros and cons of various abstractions, I have come to the realization that there is no such thing as *the* abstraction. More specifically, the appropriateness of an abstraction depends on several factors, two of which are:

1. the maintenance task being performed and
2. the application domain of the software.

My observations are elaborated in the following discussion followed by the implications.

## Observations

The most appropriate abstraction of a software system varies based upon the maintenance task being performed. This is true even when the software system is held constant. The information needed to debug a program is substantially different from that needed when introducing a new feature. If you agree that debugging is a diagnostic process often performed using a hypothesize-and-test cycle, then an abstraction that enables the verification of a hypothesis is more appropriate than one that does not. However, for making changes to a program, such as adding a new feature, an abstraction that permits "concept assignment" and impact analysis is more useful. On the other hand, if the need is to reengineer a system, as for instance when migrating from CMS-2 or JOVIAL to ADA, abstractions that enable restructuring and redesign of the code (assuming this is the intent of the reengineering task) would be preferred.

Notice that I have not said which abstraction is useful for which task, because the choice further depends upon the application domain of the software system. If a program that computes the trajectory for a surface-to-surface missile is being modified to compute the trajectory for a surface-to-air missile, the specific mathematical formulae used in the computation may be the appropriate abstractions. These formulae would also be the preferred abstractions for verifying that the code indeed computes trajectory using the formulae specified in the requirements. However, if the need is to modify the algorithm to make it more efficient, one may need entirely different abstractions (such as algorithm schemas).

The boundaries of application domain are very fuzzy, since even within the same application domain, the choice of abstraction may differ based on the specific application. For instance, mathematical formulae may be relevant in a business application computing mortgage, but not in an application computing insurance premiums, where decision charts or decision trees are more appropriate.

If different abstractions may be appropriate for applications in the same domain, it is also possible that different components (e.g., procedures or modules) of the same application (e.g., program) may best be abstracted using different notations. An obvious example is a compiler where the tokenizer may be abstracted using a regular automaton, the parser may be abstracted using BNF rules, the (non-optimized) code generator using attribute grammars, and the whole compiler using architectural notations composing the abstractions of its components.

## Implications

Let us evaluate the implications of my observations. While it is common practice for researchers and developers to tout that their tool is a panacea, it is equally common knowledge that this is very rarely true. My observations provide criteria for tool designers and tool users to assess a tool: (a) What are the specific maintenance activities the tool may be used for? and (b) What is domain of its application? It is not absolutely necessary to enumerate all the application domains since those implicitly get defined by the tool users. However, enumerating maintenance activities is not quite that straightforward. For instance, the common classification of maintenance activities as perfective, adaptive, and corrective is too coarse grained to be used for evaluating a maintenance tool.

It follows, therefore, that there is a need for classifying, at various level of details, the tasks performed by a maintenance programmer. While tasks, such as concept assignment, tracing a request, impact analysis, converting a global variable to a parameter, etc., may be found scattered in the literature, a comprehensive analysis and classification of these tasks has yet to be performed. The importance of such an analysis is evident in that it is analogous to enterprise modelling — though the latter has a much broader scope — a prerequisite to reengineering an enterprise. Brooks' identification of various roles in a Chief Programmer team is one such analysis, albeit at a macro level.

The cross-product of application domains and maintenance tasks, assuming we can classify the elements of each, is likely to be very large. One may, therefore, extrapolate that the number of relevant abstractions may also be large. While that is possible theoretically, in my opinion it would not be the case in practice. Very often the same abstraction may be useful for a large number of tasks and/or for a large number of application domains. For instance, mathematical formulae may clearly be used in scientific and engineering application. They may also be used in certain business application programs, e.g., mortgage computation.

While we await an analysis and classification of maintenance tasks, it may be prudent to identify some maintenance tasks and hypothesize abstractions suitable for them. The following is one such hypothesis.

*For activities (such as reuse and reengineering) requiring an understanding of what a software system does, abstractions that were (or should have been) used in the forward engineering of the system should also be most effective to recover by reverse engineering.*

Though not a direct implication of my observations, the hypothesis is based on them nevertheless. Notice the emphasis on *what*. It constrains the statement for maintenance activities requiring an understanding of what a software system does. (Not all maintenance activities require such knowledge.) To illustrate, the hypothesis says that, if decision charts are the most appropriate abstractions for the initial development of a system, then they should also be the most appropriate abstractions to be recovered by reverse engineering. Similarly, if an application is modelled using Finite State Machines then its implementation is best understood using Finite State Machines.

An implication of my observations and the above hypothesis is:

*Even if two programs are written in the same language, the same abstractions may not be useful in understanding them for the purpose of reuse and reengineering.*

Stated another way, it does not make sense to say that a tool helps in understanding programs in C (or for that matter FORTRAN or Pascal). This is because the abstractions used during forward engineering are dependent upon a software's application domain. For example, in telephony applications Harel's Satecharts or Petri-nets are used to model a system's behavior, in business applications ER models and decision tables are used, and in scientific and engineering applications expressions and formulae are used. Thus, neither clichés (the generic abstractions currently used by program understanding community), nor expressions, nor decision tables would be as effective as Statecharts when understanding a telephone switching system, irrespective of the implementation language. Similar statements may be made for programs in other application domains.

Where does this leave us? So far the program understanding community has focussed on recovering abstractions such as program clichés, logic or set-theoretic expressions, cross-reference databases, control flow graphs, and abstract syntax trees. These abstractions differ in the amount of information they hide as well as the effort involved in recovering them. Of these, only logic or set expressions are (to the best of my knowledge) used, by some forward engineering methods at the requirements and design level. There are several other abstractions, such as Finite State Machines, Statecharts, decision trees, decision charts, dataflow diagrams, ER diagrams, OO diagrams, SADT actigrams, to name a few, used for representing requirements or design by various systematic software development methods. If my observations, implications, and hypotheses could be generalized to represent the view of others as well, it may be prudent for the community to direct research in recovering such specific abstractions of software systems.

## References – omitted

## Biography

Dr. Arun Lakhotia is an Assistant Professor with the Center for Advanced Computer Studies and the Director of Software Research Laboratory at the University of Southwestern Louisiana, Lafayette. He received his Ph.D. from Case Western Reserve University, Cleveland, in 1989. His research interests are centered around issues related to improving the productivity of programmers and reliability of their programs. He is currently directing two projects, first to recover dataflow oriented designs and the second to recover objects from source code of legacy systems. He has contributed to research in methods and tools for developing logic programs; partial evaluation of logic programs; interprocedural flow analyses; and software metrics.