

Conceptual Programming

Steven P. Reiss
Department of Computer Science
Brown University

Abstract

We introduce conceptual programming as a process model based on our programming experiences. Conceptual programming means having programmers work directly with their conceptual models, using their own terms or languages rather than those dictated by the computer. To be useful and effective, conceptual programming must be supported by a powerful programming environment. We have developed such an environment, GARDEN, and are beginning to get feedback on its use.

1. Introduction

Our work in studying the software process has focused on understanding how our own programs are created, written, and maintained. Our environment is specialized -- an academic research/teaching environment with most of the software being written individually and relatively few programmers. But we have produced substantial systems (a workstation toolkit, a full relational database system, a program development environment, GARDEN, and most recently the FIELD programming environment) totaling at least 200,000 lines of C code over the past 6 years.

An analysis of our approaches to writing software and our thoughts on how we believe that we would like to write software resulted in a process model based on "conceptual programming" and language definition. This methodology involves providing support to allow programmers to work directly in the multi-dimensional, visual terms in which they conceptualize their systems. This allows the whole programming process, from specification to coding and maintenance, to be done in the same metaphors that were originally conceived in the design of the system. The key to making this process model effective is the ability to provide a suitable support environment. We have taken the first step toward this with the GARDEN system.

2. Conceptual Programming

Most of the various software process models agree on the general framework of how programs are put together. The program must first be defined, then a solution framework must be conceived, specified, implemented and maintained. What normally happens, however, is that the designers build a model of the solution in their heads, determining a broad framework for how the system should function. They then have to take this conceptual model and cast it into some specification language, either formal or informal. The resultant specification is then implemented in another language that the computer understands and can run efficiently. Finally, maintenance is done in terms of this implementation language while referring to the specifications. This process is further complicated by the realization that there is feedback between the various stages, i.e., while doing implementation, the specifications and even the designers' conceptual model of the system have to change.

Many of the difficulties of programming arise because this approach causes the programmers' first mental model of how the system should work to become further and further removed from the reality of the system. Neither the specification language nor the implementation language is adequate to directly reflect this model. Moreover, since programmers understand the system in their own terms, using their original model, they must continually map back and forth between this model and both the specification and the implementation languages.

Providing an executable specification or prototyping language that can be used with an advanced compiler to generate an appropriate implementation is only half of what is required. It helps by eliminating many of the complexities that are inherent to the implementation language, by restricting the number of different representations that

are used, and by allowing maintenance to be done in terms of the original specification. However, there is a considerable gap between the conceptual model and the specification language, and the multiple-representation and mapping problems still exist.

The conceptual models that programmers build for their systems are rich, multi-dimensional, visual representations that describe various aspects of the systems, for example data flow, control flow, or message traffic between various entities of the system. The representations are generally specialized to the particular problem on hand, to the particular designer, and to the solution constraints (e.g. the target architecture). Multiple representations are used to focus on different aspects of the solution (i.e. data and control flow) and to display different pieces of the solution (the overview might involve message passing while a subproblem might be understood in terms of data flow). These representations can either be nested hierarchically or can be overlapping, i.e. offering different points of view of the same solution.

Conceptual programming, the approach we are looking at for software development, attempts to allow the designer to work directly using these conceptual representations by viewing them as languages. It provides support for combining multiple languages, both visual and textual, that reflect different paradigms. It allows programmers to define new languages to match their various conceptualizations. By allowing both the syntax and the semantics of these languages to be defined, it allows the conceptual model to be used directly as the specification and the implementation language, and allows debugging and maintenance to be done directly in terms of the model.

A process model based on conceptual programming involves programmer definition of the languages that they will work with. Typically, they should be able to quickly provide at least a syntactic definition of the top-level conceptual languages they are using to design the system. This definition would allow them to build their conceptual model and to further refine it into a complete solution. The specification should eventually be formal enough to allow consistency and integrity checking, and possibly prototyping of the solution. The implementation of the solution is then done by defining the semantics of the conceptual languages so that the conceptual model can be compiled directly into an efficient representation. Thus the conceptual model becomes the program once the underlying languages are executable. Changes in the conceptual model are easily made in terms of the language they are defined in. However, the system can also be changed by minor modifications or additions to the underlying language. Note that the implementation of the language can involve other conceptual models and conceptual languages.

3. Support for Conceptual Programming

In order for such a process model to be at all usable, there must be substantial computer support for the definition and implementation of both visual and textual languages. It is essential both that programmers have a concrete representation of their conceptual model that can be manipulated, consistency-checked, and so on, and that the basic definition of the underlying languages be simple enough so that it does not get in the way of the conceptual model. Moreover, the programmers must be able to later come back and define the execution semantics of these languages so that their conceptual models can be directly executed. This constraint implies a system that can be used to rapidly develop new languages, and where programs in the new languages can be directly manipulated.

The GARDEN system¹ is our first attempt at providing this type of support. It offers an object-based model for language definition. The model provides the flexibility to easily define a wide variety of languages (we have done data flow, finite state automata, petri nets, logic programming, CSP-style message passing, and several others) using representations that are directly related to the underlying language semantics. The objects representing programs are no different from objects representing data within the system. This allows object-oriented inheritance to be used to define new languages as modifications of previously defined ones. It also allows the direct checking of the integrity of object-based programs. GARDEN also supports object evaluation so that the objects representing a program can be executed directly.

GARDEN's support for object-based programming is specialized in two ways. The first is an enhanced object model that supports object evaluation.² The model simplifies the definition of object-based languages, and offers local variables as an intrinsic part of the object model. It allows the easy combination of a variety of different languages representing different paradigms by providing a common foundation.

The second support mechanism is the set of tools that are available for defining the visual syntax of object-based programs.³ We have developed a series of packages that allow natural-looking representations to be readily constructed for all of the languages we have currently implemented (and many more we have considered only on paper). These representations can be readily adjusted by individual programmers to match their particular styles. Moreover, the resultant visual programs are directly editable using a structured editor, and they can be used for

program visualization by highlighting objects as they execute. The definition of the visual syntax is done through a direct-manipulation interface and will generally take under an hour including definition of all the editing operations.

GARDEN provides a good first step toward conceptual programming. However, it does not address all aspects of this approach. It only provides support for multiple languages in a hierarchical fashion. It does not offer direct support for very different views of the same underlying representation. Nor does it yet provide for the easy definition and use of a variety of textual syntaxes to go along with the visual ones. Finally, it only offers a prototyping environment since it is a closed, self-contained system that runs interpretively. These are all areas that we are actively researching.

4. Experience with GARDEN

We are currently working with the GARDEN system to get some sense of the effectiveness of the conceptual approach to software development. GARDEN has been used in limited contexts so far, primarily for the definition of new languages such as a visual programming-by-demonstration language based on extended logic programming with constraints, as well as the development of an environment for neural-network programming. We have also used the GARDEN system to develop several parts of itself including a system browser and more recently the start of support for a variety of textual languages. More recently, GARDEN was used to develop a simple Query-by-example system using multiple paradigms as well as a object-oriented versioning model, a truth maintenance system, a subsumption architecture system, and a marker-passing semantic net. The query-by-example system used different paradigms to reflect different phases of the program: an automata for parsing, a tree-transformation language for optimization, object-based methods for the user interface, and evaluable query trees for evaluation. The other recent efforts were student projects. These primarily took advantage of the powerful facilities provided by GARDEN and the ability to quickly prototype and edit visual representations.

References

1. Steven P. Reiss, "Working in the Garden environment for conceptual programming," *IEEE Software* 4(6) pp. 16-27 (November 1987).
2. Steven P. Reiss, "An object-oriented framework for graphical programming," pp. 189-218 in *Research directions in object-oriented programming*, ed. Peter Wegner, MIT Press (1987).
3. Steven P. Reiss and Joseph N. Pato, "Displaying program and data structures," *Proc 20th Hawaii Intl Conf System Sciences*, (January 1987).