TRENDS IN THE DEVELOPMENT OF DEVELOPMENT ENVIRONMENTS

W. Harrison

Rapporteurs: Miss J. Pennington
Miss M.J. West

Trends in the Development of Development Environments

William Harrison
IBM Thomas J. Watson Research Center
Yorktown Heights, N.Y, 10598
harrisn@ibm.com

Theory and Practice in Software Development

We must all be familiar by now with the "waterfall" simile for describing the process of developing software on a large scale. This image depicts the downhill process of development as taking place in several stages: requirements, specification, structural design, functional design, implementation, and maintenance. The stages are consecutive through time, and the information built up during earlier stages flows into the later ones. In fact, the image might perhaps have been more accurately drawn as a stairstep, however, because each phase of a project must be accompanied by significant work to lift the total information about the system to higher and higher levels.

The actual practice of software development today is less pleasant than portrayed by the theoretical waterfall simile. The information developed during each development phase must have its expression in some language, whether formal or informal. Thus requirements are expressed in a requirements language, specifications in a specification language, etc. until the implementation is finally done in the implementation language. In some cases the "language" may merely be an expected structuring of information in a document written in natural language. This is often true at the requirements, specification and design phases because the information developed up to that point is intentionally imprecise. Further precision would have required the greater effort reserved for a later phase, and the information from the earlier phase is intended as direction to a practitioner in the art in whom the designer has some confidence of his ability to continue the elaboration of the design. The use of a multiplicity of languages means that in order to make the transition from one phase to the next, the information produced in the earlier phase must be re-cast in the language to be used in the next phase. So in addition to the work required during each phase to raise the level of the system design/implementation, additional work is required to continually retranslate the designs/implementation information from language to language.

Use of the "waterfall" diagram of development also has another subtle effect. It tends to focus our attention on the succession of phases, isolating the information produced in each phase from that produced in the next. This manifests itself in the partitioning of the outputs of each phase into a sequence of sets of files: the requirements files, the specification files, etc. through the implementation files. This has the unfortunate effect of breaking the "horizontal" links that relate the consequent choices in later phases to their roots in the earlier phases. This horizontal information is generally kept, then, in the minds of the development team and lost as time passes. This horizontal information is especially useful when changes at an earlier level of abstraction are needed at a later time. Often, the changes are made only in the later documents rendering the higher-level (though earlier) documents inaccurate.

Over all, it might be better to present the "waterfall" rather as a tree, whose growth and branching represents the continuous elaboration of decisions made during the design and development process. From root requirements through specification, structuring, design and implementation, work is always directed to growth of the tree.

In fact, the "tree-growth" model can be used to illustrate more clearly the cause of the "extra effort" that shows up in today's realization of the waterfall. When we in computer science define languages for the expression of ideas, we try to make the languages "self-complete". That is, we try to make each unit of information to be expressed in the language contain a full enough description that it can stand alone. Thus programming languages tend to embed interface definition languages and system structuring languages. We do not encourage a style of design in which a program just reflects the logical function but not its name, description, or interface. When we embed this tendency in conventional software systems, it translates into requiring each file which is part of the development of a system to stand alone. Thus, what we are doing in the "waterfall" style of design is covering the tree-growth of the actual information with regions of information determined by the languages we use. The regions overlap because they must each be self-complete, and the overlap represents the extra labor that goes into re-describing the same decisions in each different language.

However, the tree-growth model of software design also suggests an alternative to the current practice of the waterfall approach to system development. We should allow the system builders to work in a continuous refinement fashion, always growing the tree of decisions toward the desired implementation. The need for periodic documentary checkpoints to mark phase boundaries, can be satisfied by way of "projection" programs that extract the relevant phase-related information for presentation and review. The early projection programs construct stylized natural language document sets, while the later ones construct the programming language source for compilation. In addition, it better accommodates the fact that in reality, a system design does not proceed evenly because not all of the components of the design can be accepted with equal confidence. If the design calls for a "table-driven-parser", professionals do not need further elaboration to accept its viability. However if the design calls for a "natural-language-recognizer", more detailed design and, perhaps implementation, work will need to be done to demonstrate that the overall design is viable. The tree-growth model allows different portions of the elaboration tree to be pursued to whatever depth is necessary to establish the soundness of the design.

Technological Requirements for a Shift in Model

In order to make the transition from the discrete refinement waterfall model to the continuous refinement tree-growth one, we must make a series of technological advances in parallel: from linear parsable languages to structured internal forms, from independent systems of files to integrated repositories, and then from flat file editors to direct structural editors.

The continuous refinement style of development embodies the gradual elaboration of material from early specification through implementation as an ever growing tree of detail. Several difficulties occur in attempting to use linear parsable languages for that purpose. In many cases, a designer or implementer will need to make an informal "sketch" of his plans before elaborating on them in detail. These plans can include diagrammatic or textual descriptive material, or a mixture of both. In any case, the design and implementation of a system contains a great mixture of linguistic concerns, from functional design through planning, performance, design points verification and testing. In addition, many different paradigms for the refinement of a problem may be employed, some problems decomposed as communicating processes, some as functional layers, some as sequential code, even some as rule-driven choices. None of these needs are addressed well by the classical model for linear parsable languages. This need not surprise us. From a historical perspective, we entered an era of large scale programming during the 1950's. Programmers needed to express complex structures in such a way that they could be executed by machines. This meant, in effect, that the machine needed to construct internally a representation of the structure in the programmer's head. However at that time, the only common medium for man-machine communication was the paper tape or the punched card, both of which are essentially linear media. Computer scientists, faced with the problem of how to pass complex structures from the minds of programmers to the internal structures in the machine through a linearizing medium, devised formal language theory. This allowed the description of rules by which a programmer could linearize his structures readably and machines could reconstruct the structures via parsing algorithms. The linearization works well for the imperative and declarative parts of the language, but works less well for the structures: process interconnection, program structuring, scoping, loops and conditional flow and commentary-related code. However, the entire motive for linearization has disappeared with the advent of display-oriented systems. We can now allow designers to work directly with projections of an interconnected structural form for the information that is shared by both the human and the machine.

The linear language paradigm served well as programmers moved from batch to interactive systems. The card decks or tapes became files within a file system. File systems, however, are very large-grain storage units; it is impossible to point reliably to elements (the individual records) within the files. This makes the retention of a variety of important connectivities difficult or impossible: connectivity within the nested elements of a structure like those which link modules within a process, connectivity along paths of decision making and refinement like those which link specification to pseudo-code and pseudo-code to implementation, and connectivities through the version history of a unit like those which show that a change from support of small tables to large was accomplished by replacing a bubble sort with a better one. We must replace the storage of program material as independent files by one which emphasizes its storage in a highly structured, integrated repository.

These goals cannot be accomplished, however, without also developing a technology for presenting and modifying the information in the structured repository. Unlike conventional file editors, or even syntax-directed editors assisting with the maintenance of a fundamentally linear representation, we need editing environments that support the presentation and modification of views of a wide variety of objects. These objects will employ diagrammatic, textual, audio or other representations and will evidence a wide variety of command functionalities.

A number of technological bases can be brought together to bear upon these transitions. Object-oriented data bases [Zdo86] provide a way of dealing with the variety of types of information while addressing performance concerns. The objected-oriented style of system structuring [Hen86] addresses the data independence and decentralized control needed for federating the separated elements of applications. Efforts defining new linguistic structures for module interconnection [Oss84] and re-use [Kru84] will yield the conceptual models needed for knitting the higher-level constructs together. Data conversion, sharing and transmission protocols [Bla86] will be used to integrate solutions implemented by different programming paradigms.

We can begin to see a new synthesis of these approaches emerging in the architecture of structure-based environments [Rei86]. In this structure, a structural shell provides the framework into which the various types of objects fit in order to federate them together to form higher level world-views. The framework provides input routing and output composition functions, information propagation mechanisms, and data repository functions. The structure-based environment will provide an open-ended structure into which new applications can be fit easily.

Impact of the Availability of New Environments

This fundamental reorganization in the data and interaction models can be expected to have a major impact on software development processes and styles of working. We can expect that over the next decade there will be increased interest and activity in:

- emphasizing reuse/rework over new construction
- emphasizing mixed-paradigm construction
- relaxing language constraints arising from linearization
- simplifying the creation of extended environments
- facilitating the maintenance of consistency

Reuse/Rework

The reuse of program material is a key to the productivity improvements needed in software in the future. The importance of simple reuse has been recognized to the extent that re-use promoting constructs like inheritence [Ing78] and generic procedures and packages [DoD83] are now appearing in many modern programming languages. However the reuse of preplanned units is only the tip of the iceberg of improvement available. The reuse of software elements by reworking to fit other similar needs has a much greater potential impact. We already employ reuse/rework schemes when we create multiple versions of a design or program for successive releases or to support user configuration choices, but the coordinated control of multiple versions will simplify our ability to manage a multiplicity of variant user audiences. We can foresee the appearance and growth of approaches to interaction in which existing designs or programs are identified as the base for the creation of higher order templates allowing sharing of a base substrate with controlled substitutions obeying axiomatic constraints. In addition, the growth of a large repository of coordinated description/refinement pairs will encourage the development of systems which search for candidate matches with descriptive material and then use the matches to suggest implementations for similar newly described problems. These matches might be made using any of a number of information-retrieval strategies, from keyword or feature matching to semantic matching driven by conceptual models [Bar85].

Mixed-paradigm Construction

The growth and availability of a library of problem/solution pairs will bring pressure to bear on allowing the reuse/rework of solutions implemented in mixed paradigms, since the identified implementations may not always match in their paradigm or language. In addition, the appearance of a framework supporting mixed-paradigm system construction will enable software builders to more easily select the paradigm which simplifies the problem at hand. The use of a graphical/diagrammatic notation for program presentation also simplifies the mixing of languages within a paradigm. (The linearization of languages has caused no end of disagreement on the choice of a least undesirable syntax.) However a system which stores program material structurally and allows the presentation form to vary from user to user can enhance the cross-readability and modifiability of programs by hiding these syntactic choices.

New Language Constructs

We will also see new exploration in the language construct arena made possible by the structural manipulation rather than the textual manipulation of design and program material [Har86]. A classic case illustrating the limits that linearization places on language choices is that of decision tables which, although widely recognized for decades as a convenient way of decomposing some complex decisions has never found its way into any programming language for lack of linearizability. However there are other examples that indicate the potential for a wide variety of new notions. Today's programming languages make use of rules determining the meaning of names that are fundamentally structural: that is, the meaning depends on the position of the name in a scoping structure. In an environment heavily supporting reuse, we might expect to see historical resolution of naming as well: that is, a name might be resolved to the meaning it was originally given, no matter how many times the software fragment containing the name has been reused in other contexts. We can also expect to see multidimensional name inheritance schemes, addressing the fact that a software fragment may be part of A, serving role B, in system context C and therefore names might be resolved in each of those dimensions even though they themselves have no nesting structure.

Simplifying Extensions

The appearance of structural frameworks for software material will also simplify the creation of system extensions which transform program material because the material will already exist in structured form. The interfaces between the environment and the extension language will need to be carefully thought out because the extension language can no longer presume that it is dealing with flat files. Use of an object-oriented methodology for interfacing object definitions with the structural framework will also simplify the creation of new environments which can be easily integrated with or interspersed into existing collections of objects. A user could, for example, add a new programming construct like a specialized loop without re-creating an entire language processor, but merely by adding a new object to some existing collection.

Consistency Maintenance

Structural environments must, by nature, provide mechanisms for communicating information among the individual constructs. These same channels can be used to introduce tighter coupling between the specification and the implementation. In fact, a description/refinement cycle easily decomposes into a description/(specification/implementation) which tightly couples the description to the specification and the specification to the algorithm. Matching not only descriptions but specifications will allow the reuse of even more material, but more importantly, when software fragments are copied, the specification and implementation are copied jointly, along with their consistency relationships. The same mechanisms that apply to re-resolving symbols and type checking are triggered to make sure that the specification axioms are valid in their new context. Furthermore, other framework-supplied facilities for management of versions and sharing will apply naturally to versions of specifications or of specification/code pairs.

[Bar85] Bartschi M., An Overview of Information Retrieval Subjects, IEEE Computer Magazine, May 1985, pp. 67-84

[Bla86] Black A., Hutchinson N., Jul E., Levy H., Carter J.L., Distribution and Abstract Types in Emerald, University of Washington TR 86-02-04, Fenruary 1986

[DoD83] go Ada Reference Manual, ANSI/MIL-STD-1815A-1983

[Har86] Harrison W., Rosenfeld J., Wang C-C., Weston B., Structured Editing with RPDE, Computer Language Vol 3 No 9, Sept 1986, pp. 93-100

[Hen86] Hendler J., Wegner P., Viewing Object-Oriented Programming as an Enhancement of Data Abstraction Methodology, Proceedings of Nineteenth Hawaii International Conference on System Sciences, January 1986, pp. 117-125

[Ing78] Ingalls D., The Smalltalk-76 Programming System, Proceedings of Fifth ACM Symposium on Principles of Programming Languages, January 1978, pp. 9-16

[Kru84] Kruskal V., Managing Multi-Version Programs with an Editor, IBM Journal of Research and Development Vol 28 No 1, January 1984, pp. 74-81

[Oss84] Ossher H., Grids: A New Program Structuring Mechanism Based on Layered Graphs, Proceedings of Eleventh ACM Symposium on Principles of Programming Languages, January 1984, pp. 11-22

[Rei86] Reiss S., An Object-Oriented Framework for Graphical Programming, Submitted for Publication

[Zdo86] Zdonick S., Wegner P., Language and Methodology for Object-Oriented Database Environments,, Proceedings of Nineteenth Hawaii International Conference on System Sciences, January 1986, pp. 378-387

DISCUSSION

Professor Habermann pointed out that on the issue of scale, abstraction from small to large problems, and thus techniques in problem solving, is difficult, and stressed that the opposite is also true. He invited the speaker to comment as to whether the model discussed would apply to small as well as medium and large objects. The speaker stated that it is necessary to draw a line where the object is complete by itself. This would probably be where the object gives the correct primitives for manipulating structure. He elaborated further suggesting that from the point of view of people and preference one doesn't want to look at each character and probably not each statement as an object, but at the best structure for communication.

Professor Henderson invited the speaker to comment on the extent commercial Smalltalk implementations adhere to the model. The speaker suggested that Smalltalk environments suffer as they only present to users a Smalltalk programming point of view, and he wanted an environment that could present all (functional, etc.) points of view.