Software workbenches: The new software development environment

by CARMA L. McCLURE
Extended Intelligence, Inc.
Chicago, Illinois

ABSTRACT

Software workbenches provide computerized assistance for the development, maintenance, and management of software systems. They differ from earlier programming environments in their breadth of coverage of the software life cycle. They are general-purpose software development environments with powerful tools for specification, design, implementation, testing, and documentation. A complete software workbench must have these characteristics:

1. A graphics interface for drawing structured diagrams
2. A central information repository for storing and managing all software system information
3. A highly integrated toolset sharing a common user interface
4. Tools to assist every phase of the life cycle
5. Prototyping tools
6. Automatic code generation from design specifications
7. Support of structured methodologies
INTRODUCTION

As new advances in hardware and software technologies have occurred, the software development environment has changed from a standalone mode in the 1950s and early 1960s to a batch mode in the late 1960s and early 1970s, and then to a timesharing mode in the late 1970s and early 1980s. In the late 1980s, a new mode based on personal workstations is becoming the preferred environment for developing software. Workstations have potentially far reaching implications for changing and improving software development. The introduction of powerful workstation tools and accompanying engineering-like structured methodologies transforms and automates the software development process.

A workstation is a complete environment including hardware and software. Its function is to provide computerized assistance for the production, maintenance, and project management of software systems. It is a personal machine dedicated to providing the maximum possible support for the individual software developer.

An extensive set of intelligent, integrated software tools called the workbench makes up the "soft part" of the workstation environment. Workbenches are tailor able to each developer's preferences and to specialized tasks such as project management, design, and maintenance; and replace traditional tools that are bound to batch processing and third generation languages of the 1960s and 1970s.

Workbenches differ from early programming environments such as UNIX and Interlisp in their breadth of coverage of the software life cycle. UNIX is a timeshared operating system in which a uniform file format is the primary means of integrating a rich array of programming tools. UNIX is a general-purpose programming environment in that it does not support a particular programming language or software development methodology. Interlisp, on the other hand, is a programming environment that supports only the programming language LISP. This gives Interlisp the advantage of tailoring and optimizing its tools for a single programming language. All Interlisp tools are written in LISP and provide tightly integrated operating systems, utility functions, editors, and debuggers.

Whereas such programming environments as UNIX and Interlisp have concentrated on tools for the coding and implementation phases, software workbenches provide powerful tools for specification, design, implementation, testing, and documentation. Workbenches are general-purpose software development environments attempting to support the full range of the software job as well as its management.

To meet its objectives of enhancing productivity and simplifying the process of software development, the software development workbench cannot simply consist of a collection of even the best tools of the 1970s and 1980s. These tools were not designed to be used in a dedicated, personal computing environment allowing customization for individual workstation owners. They were not designed to use powerful graphics capabilities to enhance the human interface. They were not designed to be used in cooperation with one another, linking together all aspects of the development process. They were not designed to capture information about an ongoing development process and evolution of a software product. Further, they were not designed as intelligent tools capable of performing many development tasks on their own.

For these reasons, a workbench providing a complete software development environment must have at least the following characteristics:

- Graphics capability
- Central information repository
- Tightly integrated tool set
- Full life cycle coverage
- Prototyping support
- Automatic code generation
- Development methodology support

GRAPHICS

When viewing a software development workstation, one is first impressed by the graphics. The ease with which objects can be made to appear, disappear, and move around the graphics screen with a mere click of a mouse button is amazing. More important, however, the better the graphics, the more productive the user interface can be.

Most people prefer pictures over words because the human mind is pictorially oriented. Narrative text is one-dimensional, but pictures are multi-dimensional, borrowing such properties as size, shape, and color from the physical world. Because the language of pictures is richer than the language of text, more information can be represented in pictures than in text, and people can grasp their meanings more quickly.

Graphical representations (or diagrams) have always played an important role in software development. Diagrams are used to define program specifications and to represent program designs. They provide the blueprint for implementing a design into code, and are an important form of software documentation.

Diagrams are really the language of software modeling because they offer a concise, unambiguous way of describing software. They are so fundamental to software analysis and design that different structured methodologies can be charac-
terized in terms of the diagramming techniques they use to model a software system.

Diagramming techniques have evolved along with programming methodologies. In the 1950s and 1960s, flowcharts were used to plan out detailed and complicated program logic. Flowcharts fell out of favor because they can give neither a high-level nor a structured view of a program. In the 1970s, structured techniques became widespread, and structured diagramming techniques such as data flow diagrams and structure charts were introduced along with them.

Although many different structured diagramming techniques are in use, an essential trio of diagramming types is needed for representing a software system:  

1. **Data flow diagram**—a friendly and familiar diagram used during analysis to define the problem components and to sketch a first, rough cut of program components and the data that pass among them (see Figure 1).

2. **Data model diagram**—a diagram used during the data modeling processing to represent the data items and the logical associations among them.

3. **Tree structure diagram**—a hierarchical diagram created during program design to define the overall architecture of a program by showing the program modules and their relationship to one another.

Minimally, a software development workbench should provide the capability of automatically drawing and updating each of the three types of diagrams. Other types of diagrams such as decision trees, finite state diagrams, and data navigation diagrams also are useful for modeling software systems.

**Beyond Automatic Drafting**

Although important in increasing productivity, a graphics capability must go beyond automatic drawing functions.

---

**Figure 1**—A data flow diagram showing the four subprocedures in the sales distribution system
There must be an underlying logical meaning associated with the graphics. This is important for several reasons.

First, when there is a meaning associated with the graphics, the correctness and completeness of the diagrams can be checked more thoroughly. Some examples are included in the Error Checking section. Second, when the meaning of the graphics symbols is captured, the same information can be represented in different but equivalent forms. For example, a data flow diagram can be automatically converted into a tree structure diagram (e.g., a structure chart) after the root is identified. Or, a tree structure diagram can be automatically converted to an action diagram. In both cases, it is the same information; only the form in which the information is represented has been changed. Thus, a workbench can accommodate the preferences of different developers and can conform to the diagramming standards of different organizations. Third, when the meaning of the symbols is stored, information necessary for automatically generating code from the diagram is captured.

Also, the logical meaning of the diagram must be stored when a diagram is drawn. Most workbench tools store the diagram or its logical meaning in some sort of dictionary or other kind of repository. This is essential to providing the capability of quickly changing and redrawing a diagram as well as automatically changing all of the affected associated diagrams. We know from past experience that this is a tedious, almost impossible task when done manually. The storage capability of a workbench is discussed further in the Central Information Repository section.

Finally, error checking of the diagrams must be automatically performed.

ERROR CHECKING

Error checking is one of the most important capabilities of workbenches. There are four basic types of error checking for diagrams.

The first type of checking is for syntax and type errors. As an example, consider the data flow diagram in Figure 1. One syntax rule for a data flow diagram is that each process bubble must have at least one data flow entering it and at least one data flow leaving it. Notice that ORDER ACCEPTANCE PROCEDURE is incorrect because there is no data flow arrow entering it.

The second type of checking is completeness and consistency checking for a diagram. Again consider the data flow diagram in Figure 1. In a completed data flow diagram, all the data flow arrows between processes are labeled with the data that is passed between the processes. Notice that in Figure 1 the data flow arrow between ORDER ACCEPTANCE PROCEDURE and FILL ORDER PROCEDURE is not labeled. A completeness check should identify this kind of missing information.

The first two types of error checking are concerned with errors in one diagram. Checking one diagram is the simplest type of error checking and the minimum expected from a software development workbench. The third type of error checking is concerned with not just one diagram but with a family of diagrams. One example of a family of diagrams is a set of leveled or layered data flow diagrams in which successive levels describe processes in increasing detail. Figure 1 identifies the ORDER ACCEPTANCE PROCEDURE and Figure 2 shows in detail what happens inside the ORDER ACCEPTANCE PROCEDURE. Consistency checking across a family of diagrams checks whether information is consistent from level to level.

Notice that the data flow diagrams in Figures 1 and 2 are not consistent because they do not show the same number of data flows going into and out of the ORDER ACCEPTANCE PROCEDURE.

The fourth type of error checking is concerned with tree structure diagrams in which functions or procedures are decomposed into more detailed subfunctions as one proceeds “down” the tree. Some structured methodologies provide guidelines for decomposing functions. For example, no function should be decomposed into itself. Workbenches should incorporate these types of refinement rules into their error checking.

CENTRAL INFORMATION REPOSITORY

Although not as visually impressive as the graphics capabilities, the central information repository is the heart of a workbench. It is the basis for integration, standardization, documentation, code generation, and reusability. No other workbench characteristic is more important.

A central information repository is a mechanism for storing and organizing all components of a software system including data structures, architectural design, process logic, screen definitions, report layouts, system diagrams, source code, test data cases, project management forms, schedules, and user documentation. A key to high productivity is getting information to developers when it is needed and in a form that is directly usable.

In some workbenches, a dictionary serves as the central information repository. However, a mere dictionary mechanism is inadequate because it does not provide information management. In other workbenches a more sophisticated mechanism called an encyclopedia is the central information repository. An encyclopedia is more than a dictionary because it coordinates and analyzes information as well as stores it. An encyclopedia is more than a database and an accompanying management system because it is a knowledge base containing facts and rules about checking the completeness and consistency of the data stored. Whereas a database and a dictionary are passive tools in which control lies with a user, the encyclopedia is an intelligent tool that can provide multiple views of information and can choose which information is to be shared. It performs a more active role in control to maintain data consistency and integrity.

INTEGRATION

Specification languages, diagramming tools, prototyping tools, dictionaries, database management systems, compilers, various types of generators, and so on are the high productiv-
ity tools of the early 1980s. However, a major obstacle to their ease of use is their tendency to be standalone tools capable of supporting only part of the software process. Usually they do not have standard interfaces to one another; and they are highly dependent upon a particular computer, operating system, and programming language. As a result, software developers must learn to use a different set of tools for each environment in which they work. Even within a single environment, developers cannot apply their knowledge of one tool to another because each tool has its own command format, specialized file structure, and range of available options.

In spite of their power, many potentially powerful tools have failed to substantially improve software productivity and quality because they do not provide integrated, continuous support to software developers in day-to-day work.

Tools integration lies at the very foundation of the software development workbench concept. It is the key to making powerful software tools practical to use. Workbench tools should interact with each other in a consistent, intuitive way and should conform to a set of well-understood standards. They should appear to a user to be cooperating with each other and aware enough of each other not to duplicate functions or messages.

There are five levels of integration. The first level, common user interface, is the minimum expected in a software development workbench. It reduces the learning curve associated
with the workbench since experience acquired using one part of the workbench can then be applied to learning other parts. A common user interface is the bridge connecting various workbench tools. When third party tools are added, individual tool differences are covered under the umbrella of a common menu system.

The second level of integration, transferability of data between tools, is another kind of bridge connecting workbench tools. When necessary, conversion routines and file transfer routines are supplied to automatically convert data into an appropriate input format for a particular tool or software package. In a smoothly running workbench, it is easy for a user to pass data from tool to tool. It is expected that the output of each tool will become the input to another.

The first two levels of integration are concerned with integrating across tools—a way of linking tools together for ease of use and ease of learning. The third level of integration is concerned with linking the phases of the software life cycle—integrating across the process of developing and maintaining software systems. In this case, the bridge between the various life cycle phases is one common representation of the system (but which allows multiple user perspectives or views), stored in a central information repository and shared among project teammates. This level of integration unites teammates, users, and management and reduces communication problems by providing one easily updated source for all information about a system.

The last two levels of integration are concerned with integrating across hardware environments—linking mainframe and micro levels, and in some cases a "middle" minicomputer level as well. The objective is to be able to perform development activities in whichever environment is most expedient. This is possible only if text, code, data, and graphics can be transferred easily between software packages and between hardware environments. In addition, some workbenches offer integration across functions—graphics, word processing, data processing, and office automation—by providing all these functions at the workstation level.

LEVEL CYCLE COVERAGE

Workbenches for general-purpose software development environments provide tools for automating the entire software life cycle with a concentration on the early life cycle phases. This front-end emphasis, or "front-end loading," of the life cycle comes from recognizing analysis and design as the most critical life cycle phases.

Specification errors can be very expensive if they are not detected and corrected in the early phases. Correcting a specification error during the maintenance phase may be 100 times more expensive than if it is corrected during the analysis phase. The completeness and correctness of the system specification affect the success of the entire software development effort. Poorly understood system requirements cause software failures. The specification is the basis for project schedules and assignments, test plans, user documentation, and program design.

Design errors often dominate software projects because of their number and the cost to correct them, especially when they are not detected early. In large projects, design errors often exceed coding errors.

More care given to design means lower-cost and more reliable systems. A system design is the blueprint for system implementation. If the blueprint does not exist or is incorrect, the system produced is probably poorly organized, poorly documented, and a nightmare to maintain.

PROTOTYPING

Prototyping tools play an important part in automating the early software life cycle phases. They are used to determine system requirements and answer questions about the behavior of the emerging system.

Screen generators, report generators, and menu builders are used mainly to prototype the user interface as a quick, friendly way of clarifying user requirements. The prototype provides users with a concrete model of how the system will look from the users' perspective. This is an effective method for identifying and correcting misunderstandings about user expectations for the system.

Fourth generation languages can be used to develop a more complete model of a system. In such cases, the prototype includes the major functions of the system but does not check for exceptions or invalid input data and does not worry about execution performance. The purpose is to give a user experience with the system by using a fairly complete model. Sometimes the model is found to be adequate enough to serve as the actual system.

Executable specification languages are the most sophisticated prototyping tools. They change system development into an iterative process whereby the system is specified and the specifications are executed to determine if the system is complete and correct. Then, based on the experience of this prototype version, the specifications are refined and reexecuted. The iterative process continues until the system is able to perform in a manner that meets all user requirements.

DEVELOPMENT METHODOLOGY SUPPORT

The software development workbench concept supports structured techniques by providing tools to automate the techniques. There are two levels to automation:

1. Automate documentation preparation
2. Automate the steps of a structured methodology.

Automating documentation preparation means providing graphics support for drawing structured diagrams such as data flow diagrams, entity relationship diagrams, state transition diagrams, and action diagrams. It also means automating the production of textual specifications such as mini-specs and pseudocode. The textual specifications are used to provide more detailed information about program procedures and data structures referenced in higher-level structured diagrams.

Since different structured techniques use different diagrams to model a software system, the types of structured techniques supported by a particular workbench will be determined by the types of structured diagrams and the notation conventions
that it offers. For example, the Yourdon Structured Design Methodology uses a structure chart derived from a data flow diagram to represent a program design; the Jackson Design Methodology uses tree structured diagrams; and the Warnier-Orr Design Methodology uses Warnier-Orr diagrams. One approach for standardizing program documentation is to restrict the diagrams and the notation conventions offered in the workbench.

The second level of methodology support, automating the process steps, means the workbench guides a user in correct use of a structured methodology. This requires that at least some level of understanding of the methodology is embedded in the tools. This could be as simple as embedded help panels that describe each step in the methodology or checklists that include the input required by and the output produced by each step. It also might include a checking mechanism to ensure that each output deliverable required by the methodology is present, correct, and complete before a user is allowed to go to the next step. In this case, a user is not simply guided through the methodology; rather, the user is forced to perform the steps in a standardized order and way. The purpose is to standardize and systematize the process of developing software.

There are two schools of thought on whether a development methodology should be embedded into the workbench. The argument for separating the methodology from the tools is that such separateness gives users flexibility in choosing the methodology or the part of a methodology that is appropriate for developing a particular system. The argument for embedding the methodology into the workbench tools is that it introduces control over the development process.

KEY WORKBENCH CONCEPTS

In summary, software development workstations differ from other powerful high-productivity tools because workstations:

1. Provide a highly interactive, responsive, and dedicated environment in which to develop software
2. Automate many software development tasks
3. Provide a pictorial view of software by means of powerful graphics
4. Enable rapid prototyping for creating models of the system to help discover and clarify user requirements
5. Collect the information necessary for automatic code generation from system analysis and design
6. Perform automatic checking to get the errors out early

These concepts enable workstations to dramatically change and improve programmer productivity.

REFERENCES