Invited Paper

Improving Software Productivity with Upper CASE Tools

KIYOSHI AGUSA*

Since software engineering is not merely a set of programming techniques, a new software engineering method should be properly evaluated and formalized before it is adopted. One recent topic in software engineering, process programming, is based on the assumption that software development can be formalized and described precisely. Without formalization, a verification method cannot guarantee that the proper work is done in a certain phase of software development. Nevertheless, human factors still remain important, especially in the early phases. Upstream CASE tools are designed for close communication among people doing related work. Technology designed to support human communication is called groupware. Although there is no definite model of human interaction, many groupware sysems have been proposed. High-performance workstations make it possible to process voices, images, and even movies. A multi-media environment contributes to giving deep sight of software as well as close communication. This paper reports Japanese research/development activities related to these current directions, focusing on upstream CASE tools.

1. Introduction

In the 20-year history of software engineering, the formalization of software processes has been a central issue. Software engineering aims to make the software development process an object of engineeering. Engineering means

> "the application of science and mathematics by which the properties of matter and the sources of energy in nature are made useful to human beings in structures, machines, products, systems, and processes." [15]

Thus it is important to find the field of science or mathematics on which software engineering is based. The downstream of the software life cycle, namely, programming, is being researched and examined as an application of mathematics. In future, we can expect several methods to support and validate activities in programming, and even automatic systems for it.

However, such a method for downstream support assumes that it is possible to capture and describe users requirements correctly. Is this a feasible assumption? The difficulty of software development will not be solved simply by effective programming. The following precepts should be observed in future software engineering [5]:

- portray systems realistically.

—View systems as systems (not as collections of subsystems).

—Recognize change as intrinsic.

- Study and preserve software artifacts.

Since a software system, especially in the case of an application program, has an interface with the end user, discernment is necessary in judging the real requirements of a wide variety of users. This may be a problem of cognitive science and may require the help of psychologists. Moreover, the analytical approach, which is a basis of science, may be inadequate and may need to be replaced with a philosophical approach. Nevertheless, we cannot wait until the results of research on human beings can provide us with fully dependable requirements before making software systems.

Our efforts to improve the productivity of software have borne fruit although we have found few fundamental principles. A software system is designed with application know-how as well as knowledge of hardware components. By accumulating experience, we have improved the productivity step by step. An example is an application-specific language called 4GL (fourth generation language). It is said that Japan has a productivity advantage in manufacturing industrial products. This achieved by collecting proposals from actual workers and acting on them, and is known as quality control (QC) circle activity. To improve productivity, it is necessary for the people involved to reach agreement on the key to creating high-quality products. Since software engineering involves a management view, a software factory has the potential for success in Japan.

Although many remarkable ideas, methods, and tools originate outside Japan, it is not always possible to adopt them as they are. Especially in the case of sup-

*Nagoya University.

port for upstream phases, there is a problem of language in addition to differences of culture. A natural (or natural-like) language interface is desirable. So that a user can feel comfortable using tools. Although the translation of terms itself is not so difficult, there are problems in graphical user interfaces, since they are usually tuned in a sophisticated manner. The sizes of Japanese fonts are larger than those of alphannumeric fonts. The display also needs to be redesigned. Moreover, the way of guiding the user may also need to be redesigned to match the Japanese way of thinking. For these reasons, many Japanese manufactures have developed their own tools rather than imported them from abroad. They gather and amalgamate their tools into an integrated software environment.

A supporting upstream environment functions to enhance collaboration among the engineers involved in a project. Groupware is a software system that is utilized for collaborative work. A target system that requires a software engineering approach is not small enough to be implemented by a single engineer, but needs a lot of manpower. Computer Supported Cooperative Work is a very hot topic. A watchful eye should be kept on CSCW, since one of its most suitable applications seems to be software development.

In the early period of software engineering history, much research was done on the problem of the lack of drawing methods for software systems. In electronic circuit design, building design, and piping design, there are established ways of drawing design results, that is, *blue prints*, with which almost all engineers can imagine the target system. Since the software is not static and is abstract, it is very difficult to prepare a good and common schematic description. The visualization of invisible software is a worth-while task improving productivity.

Many other systems superior to the ones described in this paper may exist and be in use. This paper covers only systems that have appeared in the last few years. The rather detailed descriptions occupy a good deal of space in the case of the system developed by our research group.

2. Formal Methods

It is said that a formal specification with formal verifications leads to reliable development. However, some people claim that a formal and rigorous approach can be applied only in academic research or in safety-critical systems where cost issues can be neglected. Software engineering is not a mere programming tool but a support for a well-defined and well-recognized process of software development. Once some process is defined precisely, it is an object to be automated. Since the main aim of software engineering is the improvement of software productivity, formalization is indispensable for process definition.

The automation-based paradigm proposed for software technology in the '90s is based on the following model for creating a concrete source program from informal requirements [4]:

- Formal specification
- Prototyping standard
- Specification as the prototype
- Validstion of prototype against intent
- Machine-aided implementation
- Elimination of testing
- Maintenance of formal specification
- Automatic documentation of development
- Maintenance by replay.

This model is focused on a formal specification that bridges the human-oriented process, namely, requirements analysis, and the mechanical (then automatic) process. Thus, formalization is a key to the success of software development.

2.1 Formal Approach to Software Development

The many paradigms proposed for solving problems such as incomplete requirement specification and lack of consistency checking between specification and product include the prototype paradigm, the operational specification paradigm, and the formal specification paradigm. The formal specifications are categorized into operational, algebraic, and logical approaches, corresponding to the imperative model, the functional model, and the logic model of programming, respectively.

Abstraction, localization, and information hiding are key concepts in software engineering. Modules, abstract data types, and classes are realizations of these concepts. Abstract data types have been well researched as a specification method that can be executed with term rewriting systems [12, 24].

The formal approach is not limited to university research. Joint System Development Corp. has carried out the Formal Approach to Software Environment Technology (FASET) project with the support of the Information-technology Promotion Agency of Japan [10]. The tools developed in this project are

- ASPELA (Algebraic Specification Language)
- FDSS (Functional Description Supporting System)
- DMCASE (Design Method based on Concepts CASE tool)
- STEPERS (Stepwise-Refinement Supporting tool)
- Graphtalk
- COORST (Communication Oriented Supporting tool)
- SPECPARTNER (Specification Partner).

These tools support a process for extracting formal specifications from informal requirements, which has been as the most important phase in future software development.

2.2 Formal Model of Specification

We have developed a composite requirement model, named the Requirements Frame. A requirement description based on the Requirements Frame can be translated into another type of description based on several different models, if required. It covers the data flow model, the control flow model, the relational model, and the predicate logic model. Derivation of descriptions is needed for the investigation of correctness, inconsistency, completeness, maintainability, and so on. It helps to improve the readability of descriptions by adopting the most appropriate form for a particular engineer [1, 2, 22].

2.2.1 Requirements Model

Let us assume certain requirements for a library system. To simplify the problem, we focus on the requirements for retrieving books. That is,

There exist human-type users, cards for retrieving books, and identifier numbers for all books. Cards and id numbers are data-type objects. Cards are classified into authors-cards, which are sorted by author's name in alphabetical order, and title-cards, which are sorted by title. Users can retrieve books by means of these cards.

A requirement definer first identifies objects (nouns) and object types (attributes) in a target system. He then defines operations among objects (verbs) and roles of the operations (cases), and constructs sentences bout requirements. "Cases" mean concepts about an agent, an object, or a goal of the operation [25]. A particular requirement item may be defined in several sentences.

This explanation shows that a requirement statement includes nouns and verbs as its components and that objects have roles as relations among the components. From a broader viewpoint, a requirement description includes several requirement statements as its components. From a narrower viewpoint, a noun has a type.

2.2.2 Requirements Frame

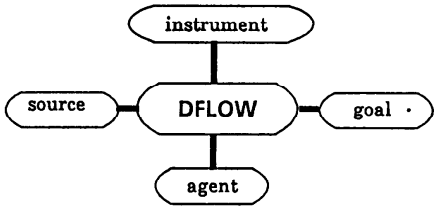
Our requirement model has been developed to allow the above structures to be represented easily. It involves several kinds of frame. The first is the *Noun Frame*, whose components are nouns and their types. Table 1 shows the Noun types provided to specify file-oriented software requirements.

A new noun appearing in a requirement description will be classified into one of the types.

The second type of frame is the *Case Frame*, whose components are nouns and verbs and cases. We provide seven different cases: *agent, goal, instrument, key, object, operation, and source.* We also provide 16 different concepts as verbs, including data flow, control flow, data creation, file manipulation, data comparison, and structure of data/file/function. Several verbs can be used to represent each of these concepts. For example, to specify the verb type *data flow*, we use *input, output, print out, display, send*, and so on. A requirement definer can use any verb as long as it can be categorized in one of the 16 concepts provided.

Table 1 Noun types in the Noun Frame.

Type	meaning
human	active object external to the target system
function	active object internal to the target system
file	passive object containing a number of instances of an information set
data	passive object containing a single information item
control	passive object for specifying control transition
device	passive object for specifying an instrument



verb	cases	entity type
DFLOW	agent	data
	source	function or human
	goal	function or human
	instrument	device

If the type of "source" is human or the type of "goal" is human, then the "instrument" case is indispensable; otherwise "instrument" should not be assigned.

Fig. 1 Case Frame of Verb, *DFLOW*.

The frame defines the case structure of a verb. For example, a *data flow* verb has agent, source, goal, and instrument cases. The agent case corresponds to a data item that is transferred from source case object to goal case object. Thus, an object assigned to the agent case should be a data-type object. An object in the source or goal case should be either a human- or a function-type object. If and only if a human-type object is assigned to the source or goal case, some instrument should be specified as a device case. This is illustrated in Fig. 1.

The Case Frame detects illegal usages of data and missing cases.

The third type of frame is the *Function Frame*, whose components are requirement sentences. Let us consider a function, "output the result of retrieval," which consists of two sub-functions: file retrieval and data output. These two sub-functions are connected to each other by the fact that the same data-type object is used in the goal case of file retrieval and in the agent case of data output. By providing "output the result of retrieval" as an indispensable function, we can detect an error when this function is absent. In the Function

Frame, there are ten essential functions, including data processing, data input, data output, file definition, and file manipulation [21]. We can check whether any of them is missing in a certain requirements description.

3. Upstream CASE tools

CASE tools are used by programmers, analysts, and designers, as well as business planners and executives at all levels, and businesse of all sizes, shapes, and structures [30]. Although there is disagreement about categorizing CASE tools into upper, middle, and lower CASE tools, since many support several phases, we will use the term *upper CASE* for tools that support upstream software development. Here, *upstream* refers to planning, describing a company's current operations, and making a standard for anew system.

3.1 Office Work and Software Development

The development of software can be regarded as a sequence of rewriting documents. Examples include rewriting a requirement description as an architectural design description, and rewriting a module specification as a source code. Tools for rewriting documents are not the same in all phases of software development. In an early phase, a tool may be very similar to an ordinaty text editor, whereas a structure-oriented editor or an editor combined with debugging environments may be preferable in a later phase. An idea will remain ambiguous until it is written down. The process of trying to write down an idea helps to clearty it. Since this is a trial-and-error process, a tool, namely an editor, should provide a powerful back-tracking facility with an acceptable speed.

3.1.1 Kanji Processing

Our model is primarily intended for documents written in Japanese. The U.S.A. and Europe have a long history of mechanized document processing with typewriters, while Japan has lagged behind because of the difficulty of processing Kanji characters. Kanji processing was hard to realize until extraordinary progress in silicon technology made it passible to handle two aspects of Kanji characters: the number of characters and number of strokes in a each character.

Since it is not feasible to provide a large keyboard with more than 6000 keys corresponding to the most frequantly used Kanji, Japanese sentences are usually entered by using a kana-kanji conversion system, in which an input kana sentence is converted into a kana-kanji composite sentence. For conversion, we need to find the syllables to pick a phrase and output Kanji and okuri-gana (kana added to a Kanji character to show its Japanese declension) successively. This is multi-phrase conversion. Since there are many homonyms in Japanese—for instance, the word *atsui* has many meanings (thick, hot, tender, serious, and so on)—and different Kanji are used to convey different meanings, not only syntactical analysis but also semantic analysis is necessary for correct conversion. Some kana-kanji conversion systems have a special dictionary, usually called an AI dictionary, for identifying the most suitable Kanji.

3.1.2 Documentation management

Once the mechanization of document processing has been achieved, it can be used for many purposes. Documents produced with upper CASE tools can be used for reference by all project members in successive phases of development. As the number of documents increases, the management of documents becomes more important. The basis of document management is a kind of data dictionary that records what kind of data is stored where, in what form, and for what purposes.

A dedicated data base for software development is called a *repository* and sometimes an *Information Resource Dictionary System* (*IRDS*). This may contain some information on constraints between two documents such as that some requirements should not be modified before some other documents have been changed. We asume the existence of such a feature for a software engineering data base [17]. Hyper TEXT is a non-sequential writing system using directed graphs, in which each node contains some amount of text or other information and is connected by a directed link with other [19]. Hypertext is becoming a popular approach for retrieving on-line documentation. Since it describes the actions needed to preserve the consistency of documents, the appropriate action will be activated when required.

3.2 Evolution Process

As mentioned before, we should *recognize change as intrinsic* and follow the cycle of requirement analysis and formal specification. The software evolues with each rotation. In other words, it is not possible to create an excellent progran without rewriting it several times. Every time we rewrite it, we can adopt new advanced technologies, fix bugs, and meet the actual requirements of users, since we learn to use it through actual operations. Users have some reaction when a target system is installed. Even though we take account of this reaction during requirement analysis, it can happen that the user behaves in an unexpected way. To make the evolution process easy, we reuse some part of the previous development. Resue should cover a whole life cycle of software, that is, reuse of the requirement specification, resue of the design process, and reuse of source codes.

A well-organized document can be expected only if the previous system was developed by using some well-recognized and well-defined methodology and associated CASE tools. The history of software engineering, especially CASE history, is not long enough for us to expect such documents. We therefore need a reverse engineering tool with which to rebuild a document of early stage of the previous development from source

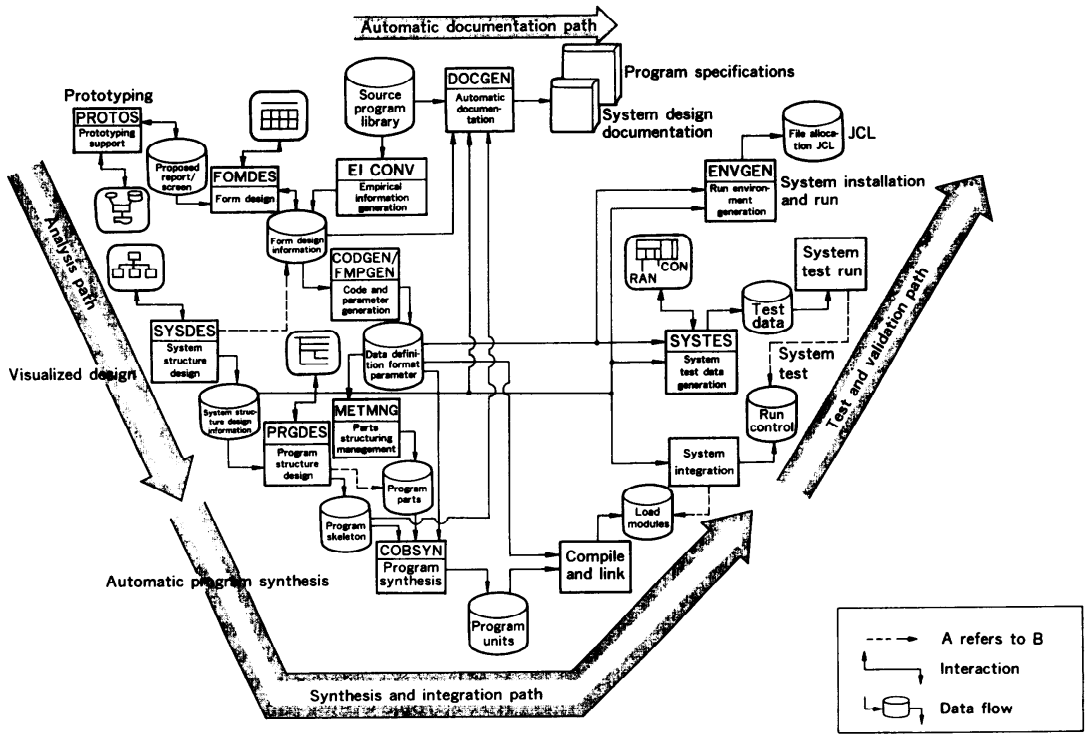


Fig. 2 Automation of OA software production by SEA/I.

codes. The word *reverse engineering* has negative implications that it helps to look for a hidden technology surreptitiously. *Reengineering* may be more suitable, since the old software is reconstructed by means of engineering.

3.3 Integrated CASE Tools

It is said that 25 to 30 percent of middle CASE specifications are transportable to lower CASE systems. Lower CASE systems generate 60 to 80 percent of program code in the system [30]. A CASE tool that covers the stages from requirement specification to coding is called an *integrated CASE* tool. Efforts are being made to standardize repositories, which are a basis for integrated CASE tools.

SEmate is an integrated CASE tool developed by Toshiba that covers the middle to downstream stages of development. A Modules Components' Connection Diagram (MCD) tool supports the design of modules by allowing an engineer to define the caller-callee relation visually. It also supports detailed design of modules by using flow charts and PAD charts. The Extended Editor (EDT) tool consists of a structure-oriented editor, a parts browser, and a syntax checker. The Software Test and Analysis (STA) tool generates a report on the complexity of modules, a cross-reference table, a test stub, and a prfile.

NEC's SEA/I (Software Engineering Architecture/One) has been used for more than five years in the field of EDP and office automation [18]. It is widely used on office computers and mainframes. SEA/I has 11 subsystems, as shown in Fig. 2. From a software specification that a user can define with reuse, the system synthesizes the full specification and automatically transforms it into executable codes.

For business applications, SEWB (Software Engineering Workbench), based on EAGLE (Effective Approach to Achieving high-Level Software Productivity), was developed to aid development with diagrams (SDF, or Structured Data Flow Diagram, and PAD, or Problem Analysis Diagram), a distributed development environment, and a seamless integrated environment [16].

In the field of communication software design, many automated development systems have appeared since the undespread acceptance of the specification language SDL (Functional Specification and Description Language). In other fields, there is no definite specification language.

4. Groupware and Negotiation

The development of a successful software system requires intelligent integration of the creative genius of imaginative and ordinary engineers. A software system involves complementary individuals such as user and developer, analyst and integrator, and director and programmer.

Groupware consists of a software tools for supporting cooperative work by team members. The collaboration is supported by sharing a common workspace to remove constraints on both time and space. The environment that makes Computer-Supported Cooperative Work (CSCW) possible is based on multimedia workstations and an intelligent network. Although CSCW is a key to effective development of not only large-scale but also small-scale software, since software evolves substantively, the state of the art of CSCW still to be at an elementary level. This means that only the hardware architecture and communication tools have ween discussed. A group model of when, what, who, and how members of a group communicate should be properly defined. It may vary from application to application. The most difficult problem of software development lies in its generality, namely, the variety of application areas.

4.1 Environment for Groupware

Recent progress in hardware technology, especially silicon technology, has made it possible to utilize ordinary communication media on a workstation. Work stations have a common work space for distributed groups. Network-transparent window systems such as X-windows, Andrew, and GMW make it possible to provide a common work space on networked workstations. Other advances in Digital Signal Processors have made it possible to utilize not only voice, but even video on a workstation.

The Team Work Station (TWS) developed at NTT's Human Interface Laboratories has a CCD camera to capture desktop images, which are overlaid on computer screens. TWS has a microphone, a headphone, and a small liquid crystal display to allow face-to-face meetings. Face windows on the computer screen can be opened if required. Beside a workstation, conventional office tools such as pencils, papers, and rulers are also used. Software products written with these devices are transferred via a CCD camera. From individual work to collaborative work, the system should bring users together smoothly. This kind of environment is called a "Seamless Shared Workspace." There are six collaboration modes in TWS, namely Tele-Screen mode, Screen-Overlay mode, Computer-Sharing mode, Tele-Desk mode, Desk-Overlay mode, and Screen-and-Desk-Overlay mode as shown in Fig. 3 [9].

The distributed multiparty desktop conferencing system named MERMAID was developed at NEC's C & C Systems Research Laboratories. It is intended to proride group collaboration support when group members work together in geographically separate locations. Like TWS, the UNIX-based EWSs used in MERMAID have a video camera, microphone, and speaker. An electronic writing pad and image scanner are attached for communication via still images and handwritten figures. EWSs are connected to each other by a high-speed data network for video. A basic ISDN line with 2B+D (B: 64 Kbps, D: 16 KBps) can be also used. A conference can be held with four modes for transferring the right to speak, or floor-passing modes. They are the designate mode, the baton mode, the FIFO mode, and the free mode. One of these is chosen before the conference by the chairperson. This system was used to develop a software system on a supercomputer with programmers who are separated by up to 60 miles. Programmers who have used it prefer ordinary communication methods such as hand drawing and voice. A more powerful scheme for computerized communication is required to make CSCW realistic.

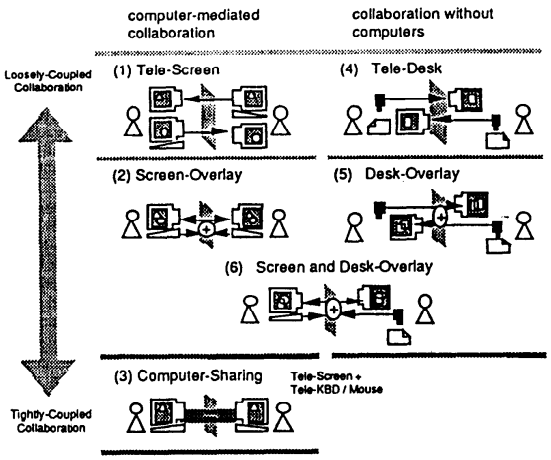


Fig. 3 Levels and Collaboration Modes in TWS Environment.

4.2 Model of Collaboration

Increasing the size of software products requires close communication among the members of a project team. This should be supported with software developed on the initative of the engineers. For this, we need a model of how engineers collaborate in software development.

The electronic secretary system is a kind of process program that defines which development tool is used and how it is used [27]. An engineer works in accordance with the task definition. The dependency of tasks and the hierarchy of subtasks are maintained by the management support system. In the description of the task, two types of communication support are provided: notification and request. The communication is in-

tended to send some message to a colleague. This is a simple electronic message asking a colleague to perform some task and waiting for the result. This is done by sending a task definition, which consists of a task procedure, a tool definition, and task result information. By sending requests, the waiting task item is added to the description of the requester's task definition. A new task is added to the task definition of the person requested, and will be refined so that it can be executed. The electronic secretary collects these communication requests, dispatches them, and records them in order to control the progress of the software development.

4.3 Negotiation and Coordination

The process for defining the specification of the target software system involves collaboration between users and developers. What kind of collaboration is the key to success? When we can divide a task of developing software into fully independent subtasks, collaboration is not needed during development. We usually set up milestones to mark stages in the progress of development, so that we can backtrack if we find that the current state is wrong. If we collaborate it is possible to work independently between two milestones. To set up a new milestone, we may need negotiation or coordination, since each subtask is executed independently. The trace of communication for negotiation plays an important role in decision making. Many alternatives are investigated and evaluated from many viewpoints. There are many tools for communication but few for negotiation.

5. Visualization

Many problems in software development are caused by its invisibility. Increasing the power of workstations makes it possible to handle multimedia and to show the extent to which the software development has progressed.

5.1 Graphic User Interface

The window system is beyond multidisplays on a single display. It should be regarded as the basis of a user interface. The window system releases a programmer from the complicated task of managing the display of overlapping windows. Nevertheless making a program with a graphical user interface is still a tedious job, since the primitive functions provided by the window system are very simple ones such as opening/closing/resizing window and selecting keyboard and mouse events. The X tool kit is a library for reusing window parts. Using standard window parts makes it possible not only to shorten the development period but also to obtain the benefits of a unified user interface.

Such tool kits should be able to handle Kanji characters. Canae, which was developed as a user interface management system at NEC, is an environment in which a programmer can easily design and implement a GUI for an application program by means of software parts and textual description. Editor pats can handle several visual media such as text (Japanese), pictures, bitmaps, graphs, tables, and hierarchies. These media are selected for CASE applications. The ratio of usage of Canae parts to a whole target CASE system is about 60 programs of CASE utilize editor parts to about 30 to 40 interface functions.

5.2 Iconic Programming

Requirements of software may vary from one user to another. Even if the requirements of a user can be represented precisely, a commonly used software may not satisfy everyone. The problem can be solved fundamentally if each user writes his own programs. One approach to this is visual programming. The User can select an icon that corresponds to a primitive function, and arrange it in the stream of a program. This type of programming is effective in CAD and simulation.

Icons can be divided into two types: data icons and function icons. Since the number of data types is restricted and since we can see the data as they exist, it is not difficult to design icons for data. However, the number of functions can increase very rapidly. Both designing and recognizing icons quickly becomes difficult to understand.

Hi-Visual 88, developed at Hiroshima University, adopts a framework in which only data icons are used to define a program. For example, the total sales-amount can be defined by using a calculator icon overlapping a sales-book icon. The result of a function, a sales-amount-report in this case, is iconized as a data icon. To show a graph of the sales amount, we place a ruler on it. These overlapping operations are continued until the desired output is obtained. A sequence of these operations is iconized for futurere use (as a secretary icon in this case) [8].

In the system sequence design method SSDESIGN, developed at Toshiba, icons are categorized into three types for human factors; Metaphor, Direct, and Ideogram. A direct icon is defined as a realistic pictogram of an actually existing obrect. Files, displays, printers, and floppy disks are examples of direct icons. A metaphoricon is defined as a pictogram that a user can understand easily as a function with a metaphor. Examples of metaphor icons are "paper and pencil" for "write" and "figure through magnifying glass" for "zooming up." Ideogram icons are abstract figures such as a circle for "correct" and an explanation mark for "caution." An icon is designed to be natural, general, useful, and to be consistent; that is, no identical icon with a different meaning can exist. Guidelines on how to make an icon are specified. One of them is syntax rules for merged icons, which are combinations of "Object" and "Action." Icons can be customized while preserving their concept [13].

Efforts in iconic programming will free end users from learning computer-oriented concepts and

mechanisms for making programs, and even for defining requirements. Like Unix commands, icons can be combined and used as a new command. This can be regarded as programming and at the same time an executable specification. It can be implemented in lower level languages if needed, just as the language C is an implementation language, whereas shell programming is a specification in UNIX.

5.3 Visual Prototyping

A famous saying rellsus that "*Seeing is Believing.*" Research is being done to make software visible.

5.3.1 Visual Environment

SSDESIGN, mentioned in the previous section, is a method for visualizing the early stage of software development. It consists of Visual Specification Description, Visual Specification Verification, and Visual Methodology for system design. The system operation specification is described by means of a diagram named SSD, which is constructed with a frame, boxes, arrow, and comments. The frame is a sheet with two coordinates: logical section and time flow. Boxes are system functions such as Action, Condition, Control, Storage, and Display. An arrow shows a flow of data or control between two boxes. The SSD Editor is a tool for defining a specification visually. The actual components of a target system are visualized by means of the SDD Image Editor, which has a data base of parts, such as switch, meter, and motor, for users' convenience. The SSDOC Visual Simulator reads the SSD to trace flows synchronizing with each other. This timing is defined by a time-axis in the SDD. The execution changes the color or shape of an image picture. This approach may be more useful in user interface design than in algorithmic problems, since concepts cannot be visualized easily.

5.3.2 Visual Rapid Prototyping

We have developed an experimental system, named VIP (VIsual Prototyping), as part of the Σ (Sigma) project. Formal requirement specification followed by software development is a key to success in obtaining a target system that satisfies users. The formality of documents sometimes requires users to learn their syntax and mathematical background. It is much preferable for users to write down their requirements in the natural languages that they use every day. In VIP, a requirement is described in a controlled natural language for communicability. It is then translated into the abstract system model from which the operation flows of each section are derived for consecutive design phases. An animation tool has been developed for verification of descriptions of requirements by users. In the case of data processing requirements for a certain warehouse, a truck carrying goods, a clerk keeping ledgers, and some related slips may appear in the animation. When a user or an analyst describes a current system, an animation shows an abstract model that extracts only the essential part of their works for data processing. How the task changes when a new software system is introduced is shown if the requirements specification description involves computerized operations. VIP is a kind of visualization tool for resolving problems caused by the invisibility of software. We have not yet implemented a function for reading a description aloud. The implementation of VIP depends heavily on the multi-media capability of a workstation.

During the design of VIP, we emphasised the following points: (1) it should be a practical tool and at the same time a pleasant tool to use; (2) it should be easy to use for personal applications; (3) it should remove tedious work related to the definition of requirements; (4) animation is effective for verification of requirement descriptions; (5) the requirements should be defined in the end users' words; (6) how to extract hidden requirements is a key to success; (7) it should be independent of the design of the software structure.

The VIP system has been implemented with three subsystems. The parser subsystem accepts a compound sentence in Japanese. Referencing the cases, it generates a set of simple parsed sentences corresponding to an opeation flow. An omitted noun and pronoun are supplemented with (1) a default noun; for instances the verb "notify" has "telephone" as a default of its tool case, (2) a related case; for instances the subjective case is regarded as the source case of "notify," (3) the same case in a compound sentence. The operation flow generator accepts the names of sections that define the column of an operation flow diagram. The symbols corresponding to a verb and its tool cases are defined in an operation flow symbol dictionary. The animation generator shows the story by using an animation dictionary that contains icons for a verb, source case, destination case, tool case, and object case. An example of a description for "transport" is shown in Fig. 4. Each section is arranged around an oval in the order of the section's column of an operation flow if the order is not specified explicitly by the user.

It is desirable that the modification should be directly applicable to an operation flow diagram and that the system should indicate the corresponding requirement description to be modified. We regard the user's description as an original work, the analyst's requirement description as a play script, and an animation system as a director. For an animation system, we should provide a capability for defining camera work in order to display, scene, and stage effects. An actor who plays a role as defined in a play script will be defined by an object-oriented model so that the definitions of actors can be reused to save time.

6. Concluding Remarks

Upstream of software development involves such difficult problems that it is not possible to find a royal road or a silver bullet. A software development system

```
display      src   obj              ;; Container appears at Warehouse
display      src   tool             ;; Track appears at Warehouse
connect      src   tool  obj        ;; Put Container on Track
move         src   dest  tool  obj  ;; Container on Track move to Shop
disconnect   dest  tool  obj        ;; Take off Container
erase        dest  tool             ;; Track disappears
erase        dest  obj              ;; Container disappears
```

Sentence: A truck *transports* a container from a ware house to a shop.

Fig. 4 Animation Description of *transport*.

is a type of software itself. To obtain a good CASE tool, a steady effort is required. An evolutionary approach to CASE may be necessary so that we can accumulate our experience and knowledge about software and related people. We have improved the productivity of software development to some extent as a result of 20 years' research. We thus have confidence that we are on the right track for the goal of software engineering.

Acknowledgement

The author is grateful to Mr. Matsumoto of NEC, Mr. Ohfude of Toshiba, Mr. Takeuchi of Oki, and Mr. Ohtsuki of Hitachi, who kindly sent internal documents used in software factories.

References

1. AGUSA, K., OHNISHI, A. and OHNO, Y. *A Verification Method for Formal Requirements Description, J. Inf. Process.*, 7, 4 (1984), 223-229.
2. AGUSA, K. and OHNO, Y. *A Supporting System for Software Maintenance-Ripple Effect Analysis of Requirements Description Modification, J. Inf. Process.*, 8, 3 (1986), 179-189.
3. AGUSA, K., OHNISHI, A., KUBO, T., NISHIYAMA, S. and IIMURA, J. *Visual Rapid Prototyping Tool-VIP, Proc. 6th Conference of Japan Software Science and Technology*, C3-3, (in Japanese) (1989), 177-180.
4. BALZER, R., CHEATHAM, T. E. and GREEN, C. *Software Technology in the 1990's: Using a New Paradigm, IEEE Computer*, (Nov. 1983), 38-45.
5. SCALING, UP. *A Research Agenda for Sofware Engineering, CACM, 33, 3, (March 1990), 281-293.*
6. DEMARCO, T. *Peoples are—Productive Project and Team*, Dorset House Publishing (1987).
7. HALL, A. *Seven Myths of Formal Methods, Software* (Sep. 1990), 11-19.
8. ICHIKAWA, T., HIRAKAWA, M. *Iconic Programming: Where to Go?, IEEE Software*, 7, 6, (1990), 63-68.
9. ISHII, H. Team Work Station. *Towards a Seamless Shared Workspace Proc. CSCW90* (Oct. 1990), 13-26.
10. *Formal Approach to Software Environment Technology*, FASET-Rep. No. 1 (Nov. 1987) and No. 2 (Jan. 1989).
11. *Introduction of Wnn+GMW*, Iwanami, 1990 (in Japanese).
12. KASAMI, T., TANIGUCHI, K., SUGIYAMA, Y. and SEKI, H. *Principles of Algebraic Language Trans. of the Institute of Electr. and Communication Eng. of Japan*, J69D, 7, (1966) (in Japanese), 1066-1074.
13. KATAO, K., KUSUI, Y., IKEMOTO, H., MATSUMURA, K. and TAYAMA, S. *Approach to System Visual Prototyping with Dynamic Icons, Proc. 1989 IEEE Workshop on Visual Languages*, (1989), 1-8.
14. KEMMERER, R. A. *Integrating Formal Methods into the Development Process, IEEE Software*, (Sep. 1990), 37-50.
15. *Langman Dictionary of the English Language.*
16. MAEZAWA, H., HAGI, Y., TSUDA, M. and INDO, I. *Perspective of Software Engineering Technology, Hitachi Hyoron*, 70, 2 (1988), 1-6.
17. MATSUMOTO, Y. and AJISAKA, T. *A Data Model in the Software Project Database Kyoto DB, Advances Softw. Sci. Tech. JSSST*, 2 (1990) (to appear).
18. *NEC Research & Development*, 96, (March 1990), 99-171 & 459-469.
19. NIELSEN, J. *The Art of Navigating through Hyper TEXT, CACM* 33, 3, (March 1990), 297-321.
20. OHNISHI, A., AGUSA, K. and OHNO, Y. *Requirements Model and Method of Requirements Definition, Proc. of COMPSAC* (1985), 26-32.
21. OHNISHI, A., AGUSA, K. and OHNO, Y. *Requirements Frame for Requirements Definition, Trans. IPS Japan*, 28, 4 (1987), 367-375 (in Japanese).
22. OHNISHI, A., AGUSA, K. and OHNO, Y. *Software Requirements Specification Technique Based on Requirements Frame, Trans. IPS Japan*, 31, 2 (1990), 175-181 (in Japanese).
23. RAMAMOORTHY, C. V. and SO, H. H. Software Requirement and Specifications: Status and Perspective, *IEEE Tutorial, Software Methodology* (1978).
24. SAKAI, M., SAKABE, T. and INAGAKI, Y. *Direct Implementation System of Algebraic Specifications of Abstract Data Types, Computer Software*, 4, 4 (1987), 16-27 (in Japanese).
25. SHANK, R. Representation and Understanding of Text, *Machine Intelligence 8*, Ellis Horwood Ltd., Cambridge (1977), 575-607.
26. *Tools Fair, IEEE Software*, 7, 3 (May 1990).
27. TAKEUCHI, A., HATANO, H. and KAJIWARA, H. *Electronic Secretary System for Software Development, OKI research report*, 57, 1 (1990), 49-54 (in Japanese).
28. TARUMI, H., REKIMOTO, J., SUGAI, M. and AKIGUCHI, C. *Experiences with Editor Parts of User-Interface Development Platform Canae, Proc. of 39th Annual Conference of IPSJ*, 2S-4 (1989), 1521-1522.
29. WATABE, K., SAKATA, S., MAENO, K. FUKUOKA, H. and OHMORI, T. Distributed Multiparty Desktop Conferencing System: MERMAID, *Proc. CSCW90* (Oct 1990), 27-38.
30. *In Depth CASE*, BYTE, April 1989.

(Received December 7, 1990)