# Notes on the Next Generation Software Factory

by

Yoshihiro MATSUMOTO

(Received September 3, 1991)

**Abstract**

Almost twenty years have passed since the first software factory started operations. From his firsthand experiences, the author introduces a typical software factory model currently being used in Japan's software factories.

A project called Japanese Software Factory of the Next Generation (JSF/NEXT), which is headed up by the author, has started to work out a new software factory model. The project aims to create an extension of current software factory models in order to meet recent needs for information system-integration and software productivity/quality improvement.

## 1. Introduction

The term "software factory" was coined from an analogy with the hardware factory. In the hardware factory there are a number of machining-shops. Each shop receives partially-made products from the preceding shop, tools from the tool-house, parts from the part-house and programs for numerical-control (NC) from the program-house. Using these tools, parts and NC programs, it adds functions, values or quality to the input products, and then transfers the machined products to the next shop. There are integral processes which collect information from every shop, analyze the behavior of each shop, and manage the process development from the aspects of quality, progress and configuration control. The activities performed inside real Japanese software factories follow a similar model, only the semantics of the machining-shop are changed to what we call "unit workload" which will be described later.

The concept of *software factory* is an overall approach in which the improvement measures are organized in order to optimize performance factors and obtain improved manifestation[8)9)] in large-scale software production. Although the word "software factory" has been recently used in a more general meaning (for example see the

Department of Information Science, Kyoto University, Kyoto 606-01, Japan

book by Cusumano[2)]), the original meaning intended by the author was the environment which allows software manufacturing organizations, housed in it, to design, program, test, ship, and maintain commercial software products in a unified manner. The software factory provides the following items:

(1) Properly designed work spaces.
(2) Software tools, user interfaces and software engineering repositories/databases (These software components are built on 'open' platforms, 'open' meaning that the platforms have public, common, portable and internationally standardized interfaces).
(3) Standardized baseline management systems for design review, inspection and configuration management.
(4) Standardized software engineering methodologies and disciplines.
(5) Education programs.
(6) Project progress management system.
(7) Cost management system.
(8) Productivity management system.
(9) Quality assurance system and standardized quality metrics.
(10) Quality circle activities.
(11) Documentation support.
(12) Resuable software libraries and maintenance support for them.
(13) Technical reference libraries.
(14) Career development system.
(15) Facilities to support clerical work.

The key plans of project management are[10)]:
(a) Project objectives are confirmed at the beginning of the project. The objectives include the target product, cost, quality level, productivity, and constraints (such as resources being allowed, standards to follow, methodology being recommended, formalities to follow, etc.).
(b) Whole-project activity is divided into many *unit workloads.* A unit workload is defined as the activity required for one person to accomplish one software configuration item.

(b.1) Unit workloads are defined phase-by-phase at the beginning of every phase. The objectives for each unit workload are derived from the project objectives.
(c) The management differs from ordinary management-by-objectives plans in the following ways:

(c.1) The objectives for each unit workload are derived by a computation called unit workload planning. Human factors are taken into account in the computation.

(c.2) The objectives for every workload are reviewed and modified in review meetings that all project members attend. The objectives are updated during the course of the project.

(d) In the course of progress, production quantity and resource expenditures are entered daily or weekly through the terminals by each person responsible for a unit workload.

(e) The system analyzes the status of progress and resource expenditures, and displays the deviation between current performance and target projections.

The objectives of each unit workload, taking into account the results of unit workload planning and unit workloads, are shown below.

(1) Workload identification.
(2) Name of responsible person.
(3) Product specifications (to be imported and exported).
(4) Estimated quantity of the software configuration item to be produced in this unit workload.
(5) Time allowed.
(6) Cost allowed.
(7) Suggested reusable items.
(8) Constraints.
(9) Resources to be used.

The objectives of each unit workload are printed individually on a form called a *unit workload order sheet* (UWOS), and are delivered to each individual assigned to that workload. (A typical form of UWOS is shown in Figure 1). The

Unit Workload Order Sheet

UW No.
UW Name:
Purpose:
Background:
Pre-Condition:
Process:
Import:
Denotation:
Export:
Post-Condition:
Constraints:
Time:
Cost:
Quality:
Size:
Performance:

Fig. 1 The form of unit workload order sheet

management system based on UWOS is characterized by the following features:
(a) Metrics and objectives for managing projects are planned at the beginning, and every member is motivated by knowing one's own individual objectives as well as project objectives.
(b) Up-to-date reports based on daily or weekly status enable members to jointly decide on the need for quick corrective action.

A project is modelled as a set of unit workloads and information flows to connecting unit workloads. The model is like a directed flow graph or a network (an example is shown in Figure 2). In order to describe the model precisely:
(1) a unit workload is represented by an *object* (a concurrent agent), and
(2) an information flow is represented by a *message passing* between objects.

The model which includes a set of objects, message passings, and the reflective computations on the concurrent objects, is called a *software factory model.*

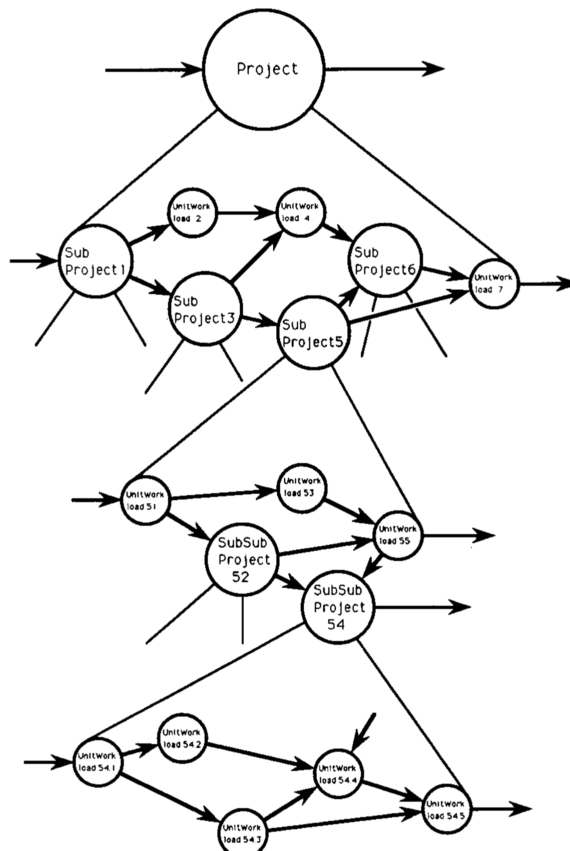


Fig. 2 An example of a unit workload network

The main part of this paper, consisting of three chapters, aims to present the very beginning results of the JSF/NEXT project:

Chapter 2: Objectives (objectives of the project JSF/NEXT are described).

Chapter 3: Basic Model (the model of a next generation software factory is described), and

Chapter 4: System Configuration (perspectives of the system configuration for constructing software factory environments are presented).

## 2. Objectives

This paper gives the very beginning concept built into the project named Japanese Software Factory of the Next Generation (JSF/NEXT). The JSF/NEXT project is a private (non-governmental) project sponsored by the Advanced Software Technology & Mechatronics Research Institute of Kyoto (ASTEM). The project is supported by many major companies including large information systems (IS) users, large computer manufacturers, IS integrators and universities. The author heads the project.

The Next Generation Software Factory is a set of concepts, basic designs and methodologies to be used in constructing future software engineering environments. It aims to accomplish the following targets:

(1) End-user orientation.

(2) Open, portable and common interfaces for IS integrators, IS users and IS manufacturers.

(3) Full utilization of the existing resources, culture and experience in Japan's software factories.

(4) Open and common platforms to ease utilization of reusable products and reusable processes.

(5) Open communication-network protoclos to enable group communication most suitable for software production.

Although Japan's current software factories have attained many goals, each factory is localized within each company, within each organization or even within each department. In addition, every factory is a closed type which means a factory environment cannot be accessed or used by outside organizations. We observed that each factory has individually established its own solid management system on which the computer-aided environments exclusively depend. This fact leads us to say that the factory is management-centered (but not technology-centered).

We have had new demands by IS users and integraters for each IS manufacturer to incorporate an open architecture into their software/information engineering

environments. One of the reasons for this demand is that international organizations of differemt countries are getting involved in the same IS projects. Another reason is the fact that members in the same project are often located in remote offices which often have different environments.

We also have received a heavy demand for the platforms to enable the users easy re-use of products/processes, efficient group-communication, and practical maintenance/re-engineering. Middleware to bridge between workbenches and targets, and the environments to combine product-centered views with process-centered views are also in demand.

Some example of what is discussed in the project are:

(1) How to promote technical transfer from software engineering researchers (SER) of universities, laboratories and research institutes to software engineering practitioners (SEP) at real production sites:

Instead of forward transfer (from SER to SEP), the main focus in western countries, we promote backward transfer (from SEP to SER). We believe the forward transfer must be managed as a part of a circular transfer which will be made possible by promoting backward transfer as well as forward transfer.

(2) How to make software engineering disciplines more open:

The nature of Japanese society is characterized by its cultural background as preferring:

· induction-centered to deduction-centered in people's thinking processes,
· bottom-up-centered to top-down-centered in the construction of things,
· "concrete-to-abstract" to "abstract-to-concrete",
· "real-practice-to-academism" to "academism-to-real-practice",
· actual experiences to abstract concepts, and
· results-after-trial-and-error to formalism which is not practiced.

We must formalize the Japanese way of operating in order to be more easily understood by all international project members.

(3) How to make environments more open:

In order to open our environments, we must define common platforms of which the interface must be common and portable. International standards must be completely followed, and if we have no such international standards to follow, we must cooperate with international bodies such as the International Organization for Standardization (ISO), to establish standards. The following standards are being studied.

(a) CCITT X.400, X.500 for group communication network,
(b) ISO/IEC JTC1 SC21 drafts on Information Resource Dictionary System (IRDS) and CASE Data Exchange Format (CDIF); ECMA Standard

149 for Portable Common Tool Environment (PCTE), and
(c) Various standard drafts delivered by ISO/IEC JTC1 SC7.

(4) How to construct common platforms for reusable items such as:
(a) Algebraic module specification,
(b) Morphism-based semantic transformation,
(c) Object-based procedure abstraction,
(d) ADT-based data abstraction and software process.

(5) How to accomplish efficient computer-supports for project coordination. Subjects included in this area are:
(a) Optimal software factory model (this might be understood as an adaptive communication model for software production),
(b) Optimal synchronous/asynchronous communication protocol (based on future communication-network such as broad-band ISDN (integrated Services Digital Network), ATM (asynchronous transfer mode), etc.),
(c) X.400/500 based group communication, and
(d) Optimal group window/shared-cursor system.

## 3. Basic Model

The major element of the software factory model is an *object,* which is a concurrent agent. An object denotes a data type which consists of variables to be bound to the values of a thing, and the methods to handle those variables. Let us assume we have thing $X$. Object $x$ represents thing $X$. We will also have [meta $x$] and [meta [meta $x$]]. Each variable of $x$, [meta $x$] and [meta [meta $x$]] is causally connected[7].

(1) [meta $x$] describes the structural and computational aspect of the denotation.
(2) [meta [meta $x$]] describes the environmental aspect of the computation and the structure (see Figure 3).

The variables denoted in an object represent the signature of a thing. A thing is a software configuration item for which one project member is responsible. A thing may be a set of data flow diagrams, state transition diagrams, specification sheets, program source texts, quality assurance sheets or group communication tracks (see Figure 4).

The real objective bodies (ROB) of these software configuration items are to be stored in each file, which will exist in a software repository. The values to which the object is bound are always consistent with its real objective bodies (see Figure 5).

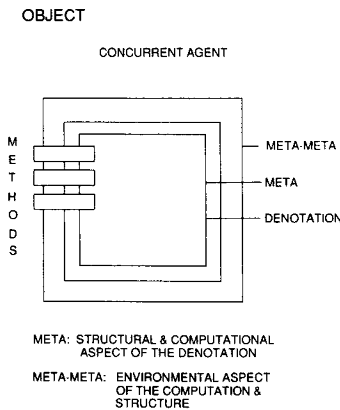


Fig. 3 Description of an object

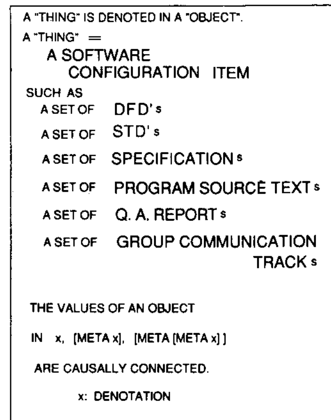


Fig. 4 Things denoted by an object

REAL OBJECTIVE BODIES (ROB's)

example:

| OBJECT | DENOTATION | ROB |
|---|---|---|
| DFD | nodes & edges | physical image |
| specification | signature equations | text |
| tool | signature | exeutable codes |
| program source | signature equations annotations | source text |

Fig. 5 Examples of Real Objective Bodies

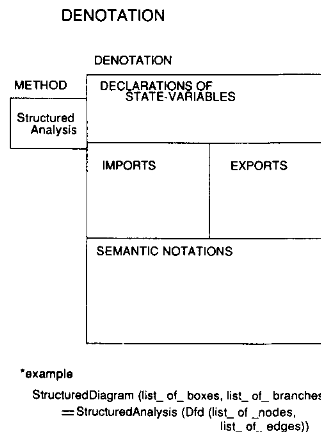


Fig. 6 Object denotation

Example 1 (Variables of an Object): In object x, which denotes a data flow diagram, the values of the variables will denote only nodes, edges and links included in the diagram. However the ROB of $x$ includes the whole physical image of the diagram.

The denotation of each object is detailed somewhat in the following paragraph. The denotation consists of variable-descriptions and method-descriptions. The variable-description includes a variable-declaration part, import part, export part and semantic notations (see Figure 6).

Example 2 (Denotations of an Object): We assume unit workload $x$ which produces a structured chart $X$. We will import the dataflow diagram which will be analyzed and converted to structured chart $X$. The import will be made by sending messages

to the object which represent the dataflow diagram. The imported items are nodes, edges and links which are included in the data flow diagram. If one needs a physical image of the data flow diagram, one can also import the ROB. Assume that object $x$ has the method (denotation) named "structured analysis". If you invoke this method, you can convert the data flow diagram to the structured chart step by step with computer-aided guidance. The boxes, branches and links in the structured chart are put in relation with the nodes, edges and connections in the data flow diagram. The relationships between boxes/branches/links and nodes/edges/ connections will be described in the part called 'semantic notations.'

Next, the missions of META (see Figure 7) are to:
(1) receive messages, and put them into a queue;
(2) take out a message from the queue;
(3) analyze and accept the message, and send the method which corresponds with the accepted message to the interpreter;
(4) interpret the method;
(5) monitor the object-status (the values of the denotation) while the denotation is interpreted;
(6) add, delete or revise denotations and/or methods;
(7) take inherited denotations and methods of other objects; and
(8) create children.

The missions of META-META are to:
(1) create and delete itself (object); and
(2) change its own dynamic state (mode) so as to adapt itself to different environments.

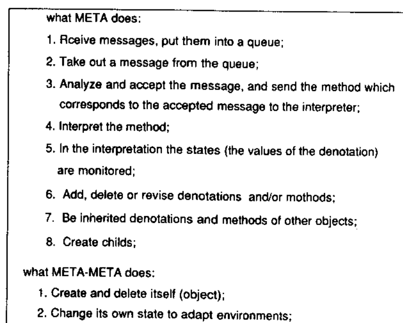


Fig. 7 Meta's applications

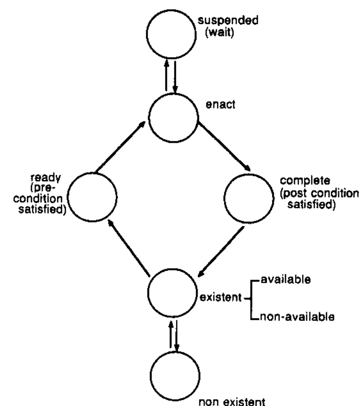


Fig. 8 Modes and transitions between modes in an object

The dynamic states (modes) of an object, shown in Figure 8, are listed below.
(1) Non-existent: The object is unknown by the system.
(2) Existent: The instance of the object is known by the system. You can lock the object so that the denotation or the ROB are not available (importable) by other objects.
(3) Ready: The object of which the precondition is true becomes ready for enaction.
(4) Enacted: The object enacted by the responsible member by the message input through a user interface is in a enacted state.
(5) Suspended: While the object is enacted, it can be suspended by the user who wants to hold the enaction.
(6) Complete: When the enaction terminates and the postcondition is satisfied, the object is in a completed state.

Now, a logical structure of *software engineering* performed to produce software products for a large information system in software factories is given below, followed by the semantics to satisfy the logical structure presented. The set of presentations on the individual software factory, which consists of logical structure, semantics and satisfaction-relationships, gives precise meanings for each software factory model.

Logical structure

The *unit workload* (UW) is the major concept used in planning software engineering projects. One unit workload represents the process of one individual implementing one software configuration item. The logical structure of the *software engineering* for a project, or the project workload, is modelled using a network in which nodes act concurrently. The network is called Unit Workload Network (UW-Net). An example of UW-Net is shown in Figure 2. A node in a UW-Net represents a unit workload. An arrow represents an event, where the event is similar to the event of LOTOS[4]. For example, UW 54.4, shown in Figure 2, is described in a LOTOS-like language in the following manner (only a partial description is shown):

```
process unit-workload-54.4 [from54.2, from54.3, from55.7, to 54.5]: noexit:=
  precondition:
    (and from54.2?54.2:scan-table, from54.3?i54.3:sensor-list,
    from55.7?i55.7:validity__range__list);
  postcondition:
    to54.5!p54.5:sensor__database__def;
endproc
```

This example means that, in unit workload 54.4, the sensor database definition is produced using the following products:

(1) The scan table produced in unit workload 54.2,

(2) The sensor list produced in unit workload 54.3, and

(3) The validity range list in unit workload 55.7.

The product produced in this process, which is the sensor database definition, is to be exported to unit workload 54.5. Unit workload 54.5 is not able to start unless the described precondition is satisfied. When workload 54.4 terminates, the described postcondition is satisfied. The symbols, from54.2, from54.3, from55.7 and to 54.5, are the names of the gates where interactions are observed.

A unit workload is represented by an object. In order to describe the idea of the unit workload, we use a language which is similar to the language called ABCL/R[14]. ABCL/R is an object-oriented concurrent language which enables us to describe meta-objects and reflective computations. An object in ABCL/R is the following:

```
[object object-name
  (state variable-declaration)
  (script
    (=>message-pattern–reply-destination-variable
         from sender-variable
       (temporary variable-declaration)
       behavior-description)
    (=>···)
    ...
  )
]
```

The state of an object is defined by a set of values to which the variables denoted in the object are bound. In a state description many variable-declarations can be included. Each variable-symbol should be bound to each state value. The statement included in "(=>···)" is called the method. In a method, the message-pattern to be acceptable, the behavior-description which describes what is to be done when the message is accepted, temporary variables to be used in the method only, and other options (such as the object names of the message-sender and the destination to which the message is to be sent) are described.

Let us assume that we have object $x$. The notation:

[meta $x$]

represents the meta of object $x$ or $\uparrow x$, while

[den $\uparrow x$]

represents the denotation of $\uparrow x$, that is $x$. For each object $x$, there exists one particular meta-object $\uparrow x$. Object $\uparrow x$ describes the structural and computational aspects of object $x$. The structural aspect is the value (state) of $\uparrow x$. The computational aspect of $x$ is described in the methods of $\uparrow x$. The state values of both $x$ and $\uparrow x$ are causally connected[7]. For each object $\uparrow x$, there exists one particular meta-meta-object $\uparrow\uparrow x$. Thus there exists an infinite tower of objects $x$, $\uparrow x$, $\uparrow\uparrow x$, .,$\uparrow(i)x$,..,$\uparrow(\infty)x$. The scope of the world to which $\uparrow(i)x$ is bound is larger than the scope to which $\uparrow(i-1)x$ is bound. But remember that the values to which all these meta's are bound are causally connected with the denotation $x$. The principle of how to design an $x$ whose denotation satisfies these conditions is now being studied in our project. This principle seems similar to *satori* in Buddhism. In our current system, we consider only $x$, $\uparrow x$, and $\uparrow\uparrow x$. In our future system, we consider a higher reflective tower.

Semantics to satisfy the logical structure

In this paragraph the semantics of the unit workoad to satisfy the logical structure of software engineering projects previously presented are covered.
Let us take unit workload $X$, and let object $x$ represent unit workload $X$. Object $x$ is bound to the states itemized below:

(1) Name

The value to which the symbol "name" is bound is used to call the object.

(2) Product

The computations performed by the methods included in object $x$ read or write the state values to which the symbol "product" is bound. The "product" represents items produced in unit workoad $x$, such as specifications, diagrams, program texts, etc.

(3) Precondition

As was explained, object $x$ becomes ready to start when the values to which the symbol "precondition" is bound becomes true. When the message to start the object arrives, a designated method is started.

(4) Postcondition

When the computation of a method in object $x$ terminates, the values to which the symbol "postcondition" is bound become true.

(5) External-awareness

This refers to the information about external objects which object $x$ is aware

of. The values to which the symbol "external-awareness" is bound are the names of other objects from which object $x$ imports product values, the names of other objects to which object $x$ sends some values, and the names of other objects from which object $x$ inherits scripts. The "external awareness" includes the awareness of the hardware environments (for example the interrupts from the devices), which will be used by [meta [meta x]]. It may also include the awareness of the human communication (in the project, organization or company), which will be used by the higher meta.

(6) Queue

The queued messages are the values to which symbol "queue" is bound.

(7) Scriptset

The values to which the symbol "scriptset" is bound are the list of message-patterns which will be used to invoke methods.

```
[object  ;;an object
  (state
        [name := the name  of myself]
        [product :=
        ;; Here the instance variables which are bound to
        ;; the values to represent a product are described.
        ;; If it is the object to denote, for example,
        ;; a network-type diagram(e.g. data-flow or state-transition),
        ;; node-id's, node-names, node-attributes, connections, arc-id's,
        ;; arc-names, arc-attributes, arc-connections, etc. are defined.
      ....])
         [precondition := an input-gate expression]
         [postcondition := an output-gate expression]
         [status :=  a current condition]
         [external-awareness := a list of informations about
                                   the external world]
         [queue := a message queue]
         [scriptset := a list of scripts]
         [evaluator := an evaluator object]
         [mode := either :existent-available, :existent-non-available,
            :suspended, :ready, :complete or :enacted)
          [system_queue := a queue to store system messages such as
                           message to change dynamic states,
                           exception-handling or handling hardware interrupt]
  (script
  (=> [:insertnode ....] ...)
  (=> [:insertarc ....] ...)
  ....
   )
]
```

Fig. 9 Description of an object representing a network-diagram

(8) Evaluator

The values to which the symbol "evaluator" is bound are the names of the evaluator objects in which method denotations are evaluated.

(9) Mode

The values to which the symbol "mode" is bound are dynamic states which are shown in Figure 8.

(10) Status

The values to which the symbol "status" is bound are temporal measures such as the time spent to finish a product, the time spent to complete a method, the number of items included in the current product, the current resource consumption, etc. The values also include constraints which characterize the object.

In addition, the values to which the states of *x*, [meta *x*] and [meta [meta *x*]] are bound are causally connected. The concepts of *x*, [meta *x*] and [meta [meta *x*]] are summarized in Figure 9. Figure 10 and Figure 11, respectively. In Figure 9,

```
[object  ;;a meta-object
  (state ;;causally connected
  (script
    (=> [:message :Message Reply-Dest Sender]
        [queue := (enqueue queue [Message Reply-Dest Sender])]
        (if (and (eq mode ':ready) (eq (eval precondition) true) then
             [mode := ':active]
             [Me <= [:begin]]))
    (=> [:begin]
        (temporary  mrs scr newenv [object := Me]) ;;declare temporary
                                   ;;variables local to this method only
        [mrs := (first queue)]
        [queue := (dequeue queue)]
        [scr := (find-script (first mrs) scriptset)]
        (if scr then
          [newenv := [env-gen <== [:new (script-alist mrs scr) state]]]
          [evaluator <= [:do-prg (scr$body scr) newenv [den Me]] @
                        [cont ignore  ;;the values evaluated are ignored.
                         [object <= :end]]]
          else (push status 'Cannot accept the message' (first mrs))
            [[meta [meta Me]] <= [:exception Me Me]]
            [Me <= [:end]]))
    (=> [:end]
        (if (not (empty? queue)) then
            [Me <= [:begin]]
         else
            [mode := :'ready]))
    (=> .......)
    ....
]
```

Fig. 10 Description of a meta object

```
[object  ::a meta-meta-object
  (state ::causally connected
  (script
    (=> [:system_message :Message Reply-Dest Sender]....
          ;;If the object receives messages from the hardware or other
          ;;objects which are superior to myself(the messages indicate
          ;;object-creation/deletion, needs for mode-change, etc.),
          ;;the object initializes states, and control the interpretation
          ;;of denotations.
    )
  )
]
```

Fig. 11 Description of a meta meta object

object $x$ which represents a network diagram, e.g. a data-flow-diagram, or a state-transition-diagram, is partially shown. Meta-object, partially described in Figure 10, describes the structural and computational aspect. It describes that a message which has arrived is put into the message queue, and processed by the evaluator which interprets the denotation of object $x$.

Meta-meta-object, shown in Figure 11, describes the creation and communication aspects. It describes how the object is created, how it interacts with signals sent from the hardware and how it communicates with other external objects. Messages which cannot be interpreted by [meta $x$] or $x$ are handled by [meta [meta $x$]] through communications with external objects.

## 4. System Configuration

The major components which constitute a Next Generation Software Factory are summarized in Figure 12. The functions of each component are described below.

(1) User interface: This symbolizes the CRT and the keyboard of the workstation. The workstation has a permanent memory (disc). The objects for which the owner of the workstation is responsible are stored. The workstation has a work space in which the necessary objects are fetched in and object-computation is performed (see Figure 13).

(2) Unit workload (object): A composite object represents a unit workload. The object composition is three-layered. The innermost object represents a denotation. The second-layer object describes the meta of the innermost object. The outmost object is the meta of the second-layer.

Example 3 (a Unit Workload Object): Let us assume that we have a unit workload to produce a source program text "XYZ". The source program text, which are

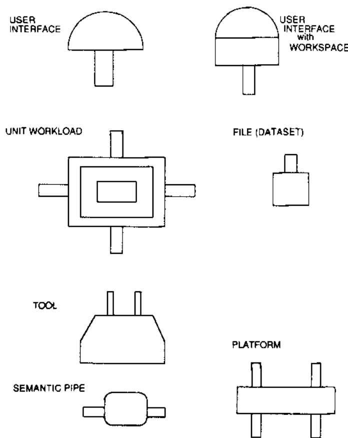


Fig. 12 Elements of environments

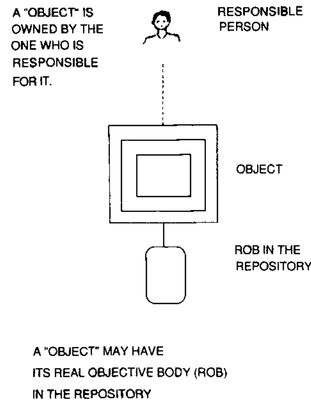


Fig. 13 Elementary configuration

the ROB of the object, will be stored in a file (in the repository). However, major items included in the file are selected and to these values the variables of product-object "xyz" are bound. The tools to view both these values and the ROB, to maintain consistency between these values and the content of the ROB, and to maintain consistency between the values of the different objects, are available from inside the method capsulated in object "xyz".

(3) Tool: A tool is an object which imports a file (or a dataset), processes it and exports a new file. A tool itself is also an object, where the ROB of the tool (executable codes) are stored in a file (in the repository).

(4) File (or dataset): A file in the software engineering repository stores one ROB.

(5) Semantic pipe: Sets of data to be transferred between user interfaces, tools, objects and files are transformed from some syntactic-forms to other syntactic-forms without changing semantics, using semantic pipes. The semantic pipe also is an object.

(6) Platform: Through the platform many individuals who are located in remote offices can exchange datasets, use tools, and exchange messages in the same project. The platform may include communication networks.

Unit workloads (objects), tools, semantic pipes and files (in the repository) may be connected in various ways through the platform. A basic configuration is shown in Figure 14. The user may access a unit workload, bring it to one's own work space, and produce products using tools. The produced products are saved in the file. (Alternative configurations are shown in Figures 15 and 16.)

A functional prototype of object management in the Next Generation Software Factory has been experimented with through the research and development of the

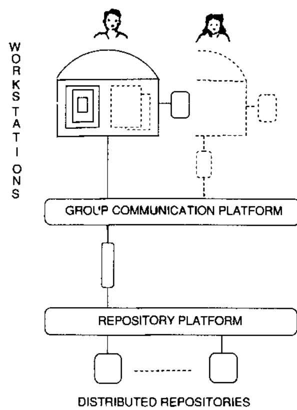


Fig. 14 Typical configuration

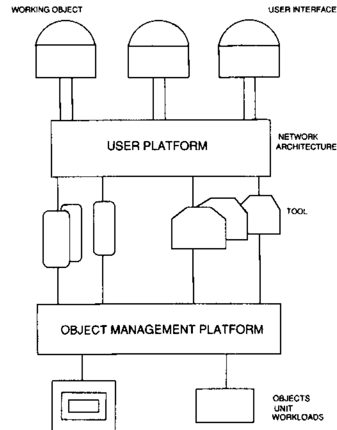


Fig. 15 Alternative configuration

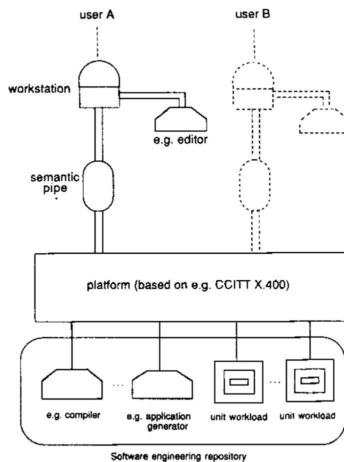


Fig. 16 Another alternative configuration

Kyoto University Software Project Database (KyotoDB)[11], which has been developed in the Department of Information Science, Kyoto University. A comparison of the data models applied in KyotoDB, with those in PCTE[3], CAIS[12], ATIS[1] and AD/Cycle™[5] is provided here.

PCTE applies to an entity-relationship model where each entity type represents a product (software configuration item), a process, a tool, a dataset, and a person type, etc. Each relationship type in the model represents a structural or computational aspect of the link connecting entity types. A set of interconnected objects, links, relationships and attribute types can be defined in the schema definition

sets. PCTE provides a public tool interface that can be used as a portability interface and integration support for executing tools and programs defined in the data model in distributed environments.

In KyotoDB major objects are of the type called unit workload. The object encapsulates the values which are associated with the product produced in the unit workload and the methods to create, revise and delete these values. A unit workload object communicates with user interface, tools, files to store the product, and other unit workload objects. Products and tools are stored in a repository which is seperate. The unit workload objects in KyotoDB communicate with each product-file or tool stored in the repository which is designed based on CDIF. Version management is made based on the unit workload network. If any revision of the product produced in a unit workload (which corresponds to a node in the unit workload network) is made, the side effect caused by the revision is tracked using branches which connects other nodes to it. A composite use of tools is programmed in each method capsulated in the unit workload objects. A complicated dialogue-input from the user interface is analyzed and processed in each method capsulated in the same object. These functions of KyotoDB are quite different from those in the PCTE. However, reading the purposes of the Pact activities[13] has led us to believe that the purpose of the data model served by KyotoDB is compatible with that of the common-services layer of the Pact environment's architecture (which is the second layer next to the PCTE core). This belief led us to start a study on the possibility of connecting KyotoDB to the PCTE core, although it has not been completed yet. CAIS and ATIS look similar to the PCTE core from our viewpoint of the KyotoDB.

The repository management in the AD/Cycle has the means to manage specifications and their execution in conceptual, logical and storage views. It allows the selection of integrity, security, trigger and derivation policies. The major element in implementing these views and policies is the object. The scope which is covered by the repository management of the AD/Cycle is approximately the same as the scope of KyotoDB. However, communications between objects (unit workloads) in KyotoDB are implemented by the reflective computations between objects, while Repository Manager™ serves as a cooperative control between objects in the AD/Cycle. The reflective computation, which is applied in KyotoDB, is useful for distributing objects.

(AD/Cycle™ and Repository Manager™ are trademarks of International Business Machines Corporation.)

## 5. Conclusion

The major purpose of this paper was to introduce an early report on the JSF/NEXT project. The paper also described the basic element of KyotoDB (a prototype software engineering project data base) whose development is a main part of the project effort. The Next Generation Software Factory, which will be the result of the project JSF/NEXT, aims to provide the concept of a software/IS-engineering environment which has portable, common and open interfaces. The Next Generation Software Factory is based on a software factor model which has long been practiced in the Japanese software factories.

**Acknowledgements**

The author would like to thank Dr. Yutaka Ohno, President of the Advanced Software Technology & Mechatronics Research Institute of Kyoto, for his support for the JSF/NEXT project. The author would also like to thank Mr. Jeffrey Mauro (from the U.S.A.), a visiting student at Kyoto University, for reviewing the English of this paper.

**References**

1) ANSI X3H4 working draft, A Tool Interface Standard, ANSI X3H4/90–187 (1990)
2) Cusumano, M., Japan's Software Factories, Oxford University Press (1990)
3) The ECMA Standard 149, The Portable Common Tool Environment, ECMA (1991)
4) van Eijk, P.H.J., C.A. Vissers and M. Diaz (ed.), The Formal Description Technique LOTOS, Elsevier Amsterdam (1989)
5) Hoffnagle, G.F. (ed.), IBM Systems Journal, Vol. 29, No. 2 (1990)
6) International Organization for Standardization, Information Resource Dictionary System(IRDS), Services Interface, Working Draft, Revision 11, ISO/IEC JTC1/SC21 N4895 (1990)
7) Maes, P., Concepts and Experiments in Computational Reflection, Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), pp. 147–155 (1987)
8) Matsumoto, Y. et al., A Softwar Factory, in "Software Engineering Environments", North-Holland New York (1981)
9) Matsumoto, Y., Software Factory: An Overall Approach to Software Production, in "Software Reusability", IEEE Publication (Cat. No. EH0256–8) (1987)
10) Matsumoto, Y., Approaching Productivity and Quality in Software Production-How to Manage a Software Factory, Proc. DRP '87 International Conf., pp. 103–122, Diebold Deutschland (1987)
11) Matsumoto, Y. and T. Ajisaka, A Data Model in the Software Project Database KyotoDB, in "Advances in Software Science and Technology", Vol. 2, Iwanami Shoten Tokyo (1990)
12) Obernsdorf, P.A., The Common Ada Programming Support Environment (APSE) Interface Set (CAIS), IEEE Trans. on Software Engineering, Vol. 14, No. 6, pp. 742–748 (1988)
13) Thomas, I., Tool Integration in the Pact Environment, Proc. 11th International Conf. on Software Engineering, pp. 13–22 (1989)
14) Watanabe, T. and A. Yonezawa, Reflection in an Object-oriented Language, Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), pp. 306–315 (1988)