ISHYS

Designing an Intelligent Software Hypertext System

Pankaj K. Garg and Walt Scacchi

University of Southern California

L arge-scale software systems are developed, used, and maintained by teams of people cooperating over long periods of time — a cooperative phenomenon often called the "software life cycle." A typical software life cycle is error prone, expensive, and unpredictable. Other researchers have discussed these problems at length elsewhere, as in Frederick Brooks' insightful article on "Essence and Accidents of Software Engineering."[1] We agree with Brooks that no simple method exists for overcoming the many problems of software engineering.

However, we believe that careful research into factors influencing the software life cycle will help solve these problems.

The web of computing. We seek to gather, into a unified framework, knowledge regarding the effects of various circumstances influencing the software life cycle. In turn, this unified framework will provide an experimental basis for exploring solutions to some hard problems in software engineering. To describe this framework, Kling and Scacchi have coined the term "web of computing."[2] The following

three interrelated aspects of the software life cycle provide a basis for this web:

(1) **Tangible products produced and used throughout the software life cycle;**
(2) **Settings in which software systems are developed, used, and maintained;** and
(3) **Processes carried out during the software life cycle.**

As Figure 1 depicts, these aspects in the web of computing are mutually influential and co-evolutionary. For example, the fact that an office information system was developed in a particular manner influences work patterns of office workers using that system. Conversely, user work patterns influence the system's ongoing development and evolution.[2,3] We know of one such case in which office workers assumed that a flaw in an enrollment system's design was one of the system's many accounting rules.

Similarly, the expertise of individuals developing a system may relate directly to the number of bugs discovered later in the system. In practice, many such situations arise in the software life cycle. Unfortunately, no unified theoretical framework exists that would help to predict such phenomena and to understand exact relationships between products, processes, and settings of the software life cycle.

Ultimately, we want to build a knowledge-based project support system that (1) embodies relationships between products, settings, and processes of the software life cycle, and (2) provides assistance to its users based on the system's understanding of these relationships. Building upon our previous work in software hypertext systems,[4] this article describes the architecture of Ishys — an intelligent software hypertext system — and considers a limited number of the web of computing's different attributes and aspects.

Developing and using software hypertext systems. Hypertext, an information storage structure, can be viewed as a graph with nodes containing information units and edges (links) that explicate relationships between information units. Different nodes can contain different types of information, including source codes of programs, natural language text, figures, and graphs.

Chief hypertext characteristics are the availability of

- **Hierarchical and nonhierarchical linked nodes;**
- **Typed links (defined relations) between nodes;**
- **User-defined attribute/value pairs for nodes or links;**
- **Paths of linked nodes that can be treated as persistent objects;**
- **Nodes with more than one version;**
- **Procedures attached to nodes and links that are activated when a node or link is visited;**
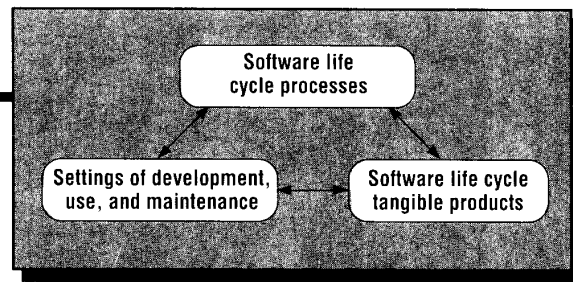- **Editing abilities for the nodes' information content;**

Figure 1. Aspects of the web of computing.

- **Support for concurrent users;**
- **Graphics in nodes;** and
- **Graphical browsing of linked node networks.**

Software hypertext is hypertext in which nodes contain and organize information pertaining to the development, use, and maintenance of a software system. Systems supporting the management of software hypertext, in conjunction with various software engineering tools, can provide an integrated software engineering environment. The advantage of this combination (a hypertext system plus software engineering tools) is that one can exploit the facilities of tools for automated information processing while using hypertext system facilities for storing and retrieving information. We have designed and implemented such a hypertext-based software engineering environment called DIF,[4] the documents integration facility at USC's System Factory — an eight-year-old project that conducts experiments in researching, developing, using, and maintaining large-scale software systems.[5]

DIF supports not only the hypertext capabilities mentioned above, but also provides

- **Consistency and completeness checking for formalized software document nodes;**
- **Intra- and interdocument traceability;**
- **Template-based document standards;**
- **Multiversion documents with or without shareable annotations;**
- **On-line software document inspections and walkthroughs;** and
- **Implicit software process modeling.**

Intelligent software hypertext systems. DIF, a practical but passive system, contains little explicit knowledge about its surrounding environment. We will describe the design of Ishys, an extension to DIF that incorporates and manipulates this kind of knowledge. Ishys, coupling DIF with knowledge of its surrounding web of computing, not only subsumes DIF's capabilities but does so in ways that make Ishys an active agent supporting the roles of other agents and participating in the software life cycle. Ishys' intelligent behavior springs from its ability to (1) automatically derive relationships between hypertext nodes, (2) automatically determine attributes of hypertext nodes, and (3) coordinate and schedule agent tasks in the software life cycle.
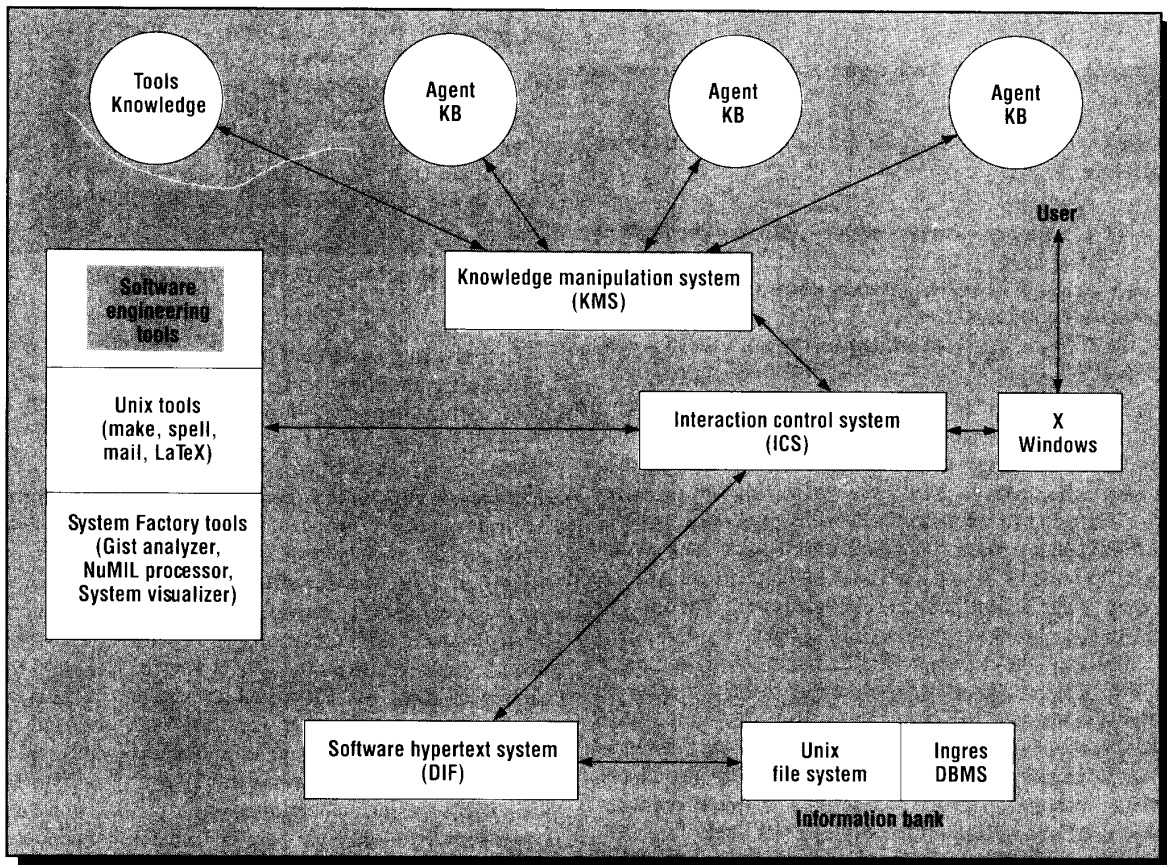
Figure 2. The Ishys architecture.

We designed Ishys as a distributed knowledge-based system (see Figure 2). It has four chief components:

(1) A knowledge manipulation system with access to different pockets of knowledge for each agent about which Ishys knows;

(2) A software hypertext system that maintains tangible products of the software life cycle;

(3) Software engineering tools used to engineer and manage information contained in the software hypertext; and

(4) An interaction control system that interacts with users and controls user access to the knowledge maintenance system, the software hypertext system, and software engineering tools.

In typical Ishys scenarios, users enter Ishys and declare themselves to be agents playing particular roles about which Ishys knows. Ishys invokes appropriate knowledge bases that must become active as a result. The knowledge maintenance system then advises the interaction control system what is expected of (and how to help) the current user.

Based on the user's actions, Ishys can appropriately modify different pockets of knowledge. Actions that users perform through Ishys operate on software hypertext nodes and links. This can require the invocation of one of the software tools interfaced with Ishys.

Ishys' knowledge base follows directly from our web-of-computing model. We will initially consider selected attributes of this web to limit our effort while still providing desirable capabilities. These attributes have proven particularly salient in recent empirical studies of software development projects.[1,6] Figure 3 shows relationships between the web of computing's three aspects and how they are modeled in Ishys (as listed below):

(1) Products are modeled in Ishys as a software hypertext, a network of semistructured software object descriptions. Using the knowledge maintenance system. Ishys intelligently maintains nodal attributes and relationships.

(2) The setting of development, use, and maintenance is captured primarily by representing a taxonomy of the various roles that participating agents in the software life
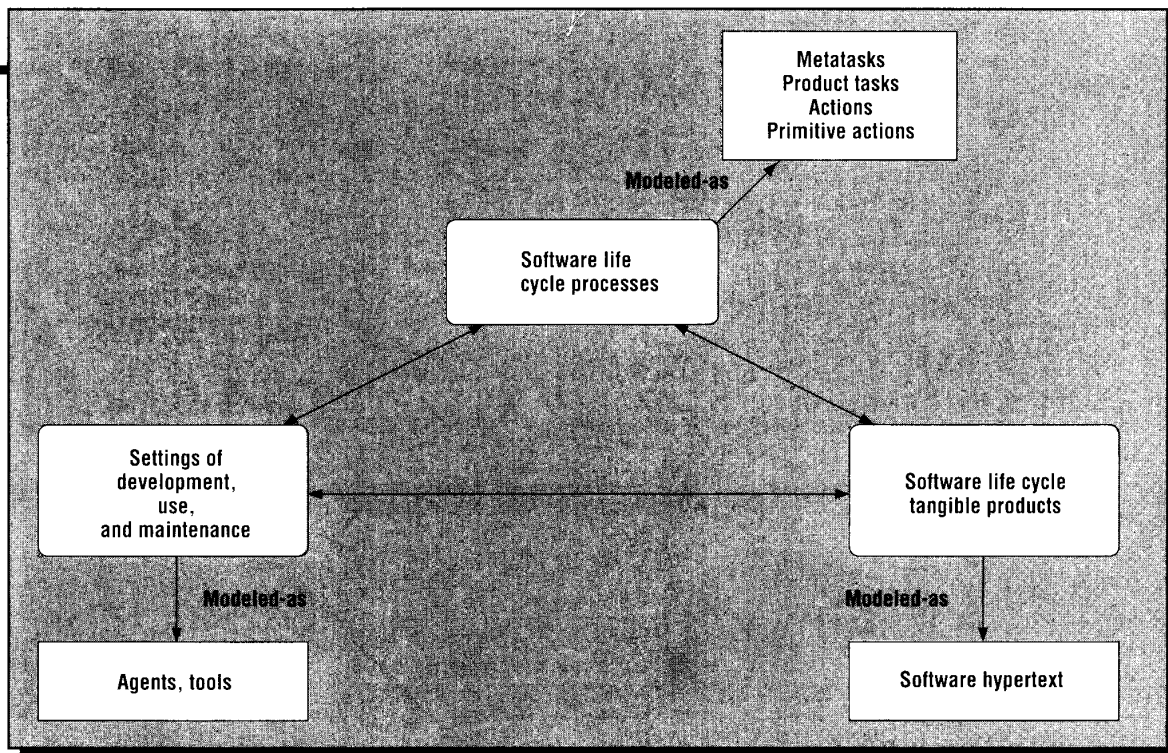
Figure 3. The Ishys knowledge base.

cycle play, and knowledge about the various software tools available in the setting. The taxonomy of roles helps us develop an understanding of the skill requirements of different agents in the software life cycle. Agents playing different roles perform specific tasks on different types of software objects. Each agent has a set of characteristics maintained in the knowledge base for the agent. Knowledge about the various software tools available in the setting enables Ishys to suggest appropriate tools for specific tasks, make sure that application preconditions have been satisfied, suggest options for using a tool in a specific instance, and associate certain tools with specific agent roles.

(3) Ishys models the process of producing software (the software life cycle) as a set of relationships between tasks of various agents in the system. We consider two kinds of tasks — metatasks and product tasks. Metatasks construct product tasks and relationships between product tasks. Product tasks modify the information content of at least one node in software hypertext. Either kind of task may require searching or navigating through software hypertext. Tasks are composed of actions that, in turn, are composed of primitive actions. Actions denote processing steps performed by an agent (with or without an appropriate tool) to produce or modify software product information. Primitive actions are Ishys commands invoked by agents to perform each step of an action. With this three-level approach, software process knowledge (the structure of software engineering tasks) is conveniently mapped into a set of system commands.

Let's examine in further detail how individual components of Ishys' knowledge base are constructed. We will subsequently describe some novel applications that potentially can be supported through Ishys.

Tangible products. The software life cycle's tangible products are descriptions of the target software system from different perspectives. The requirements document describes the system from a user's perspective. The functional specifications document describes the system from a behavioral perspective. The architectural design document describes the system from a structural perspective. The detailed component design document describes the system from an algorithmic perspective. The source code document describes the system from a computational perspective. The testing document describes the system from a validation perspective. The user's manual describes the system from a developer's perspective. And the maintainer's manual describes the system from an evolutionary perspective. By comparatively analyzing these perspectives, users can acquire a deeper understanding of what a software system does, how it does it, and why.

Two issues are involved in representing the information of these perspectives: (1) specification of the structure of descriptions must be maintained, and (2) diverse types of information must be maintained. DIF, our software hypertext system, deals effectively with these two issues through concepts of Forms and Basic Templates (BTs).
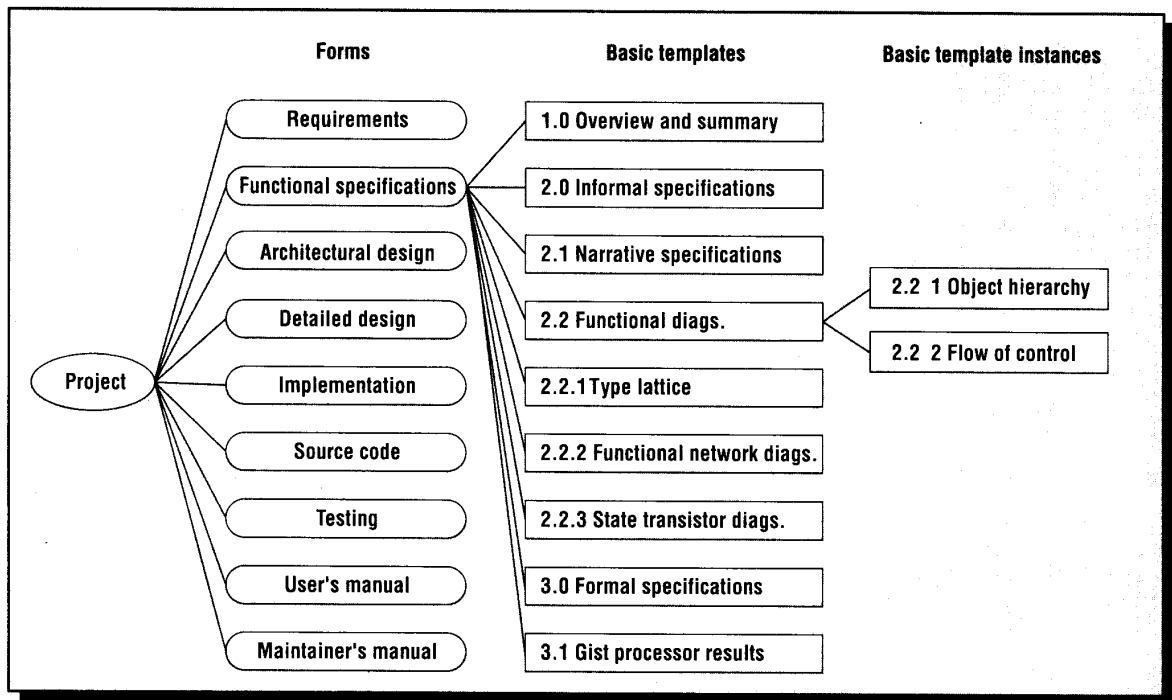
Figure 4. Hypertext of software documents in DIF.

BTs are hypertext nodes containing semistructured information. The extent of a node's "structuredness" is indicated by its type and attribute values (its syntax and semantics). Forms are tree-structured organizations of BTs. Figure 4 shows the Form and BTs that were used for System Factory functional-specification descriptions. Forms provide a way of specifying what needs to be documented in the software life cycle, while BTs contain parts of documents. Attributes attached to BTs provide associative retrieval of information. Links attached to BTs provide a means of browsing through software hypertext. Our references detail issues involved in maintaining software life cycle documents in a hypertext.[4]

Settings of development, use, and maintenance. One part of a software system's setting of development, use, and maintenance is captured through the notion of agents that Ishys supports. A second part of the software life cycle's setting is captured by representing information regarding the software tools accessible in the setting. Accordingly, when performing tasks, agents employ automated tools to create or modify software objects and relations.

Agent roles. Agents are people or intelligent systems that play well-defined roles. An individual in the software process can play the role of more than one agent. For example, a person who has designed, implemented, and used a personal database management system is at different times a designer, implementer, manager, and user. The following describes four categories of roles, as used in the KBSA (knowledge-based software assistant) project:[8]

(1) **Users** — Agents who will use the target software system (end users) and additional agents who want the system developed (clients).

(2) **Managers**—Agents responsible for software project management. We will consider two managerial agent roles: agents who coordinate the activities of other agents in the process (the process manager, or PM) and agents who analyze and evaluate the status of the process (the quality assurance manager, or QAM).

(3) **Developers**—Agents who develop the system. Their tasks involve design and innovation, in which they transform user requirements into a working target system. Developer agents play roles including Analyst, Specifier, Designer, Implementer, Tester, Integrator, and Technical Writer.

(4) **Maintainers**—Software systems are often modified to handle changes in the requirements or discovery of system development errors. Maintainers make such modifications, but may not have participated as initial system developers.

The preceding taxonomy, reflected in Figure 5, helps us conceptualize task skills of agents in the software process. Each role provides the basis for at least one major task in a typical software life cycle, but does not preclude the existence of interactions between agents (necessary when agents have intertwined tasks).

However, this does not mean that every instance of a software life cycle must contain each type of role and corresponding task. It means that, if a task of a particular nature is to be accomplished, it must be performed by an agent skilled at performing the roles required by that task.
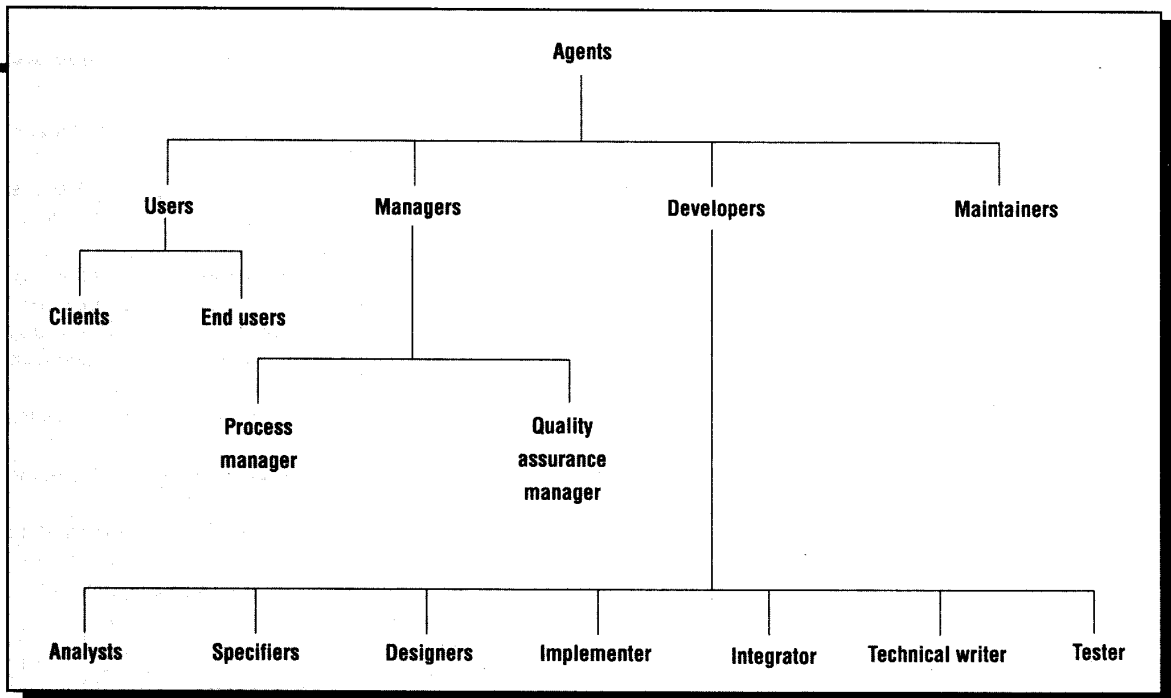
Figure 5. A taxonomy of agent roles.

Tools. Tools are objects that manipulate hypertext BTs by changing BT attributes. For example, entering information in a BT with the use of an editing tool changes the Edited attribute from No to Yes. Similarly, compiling a BT containing source code changes the Compiled attribute from No to Yes. The pre- and postconditions of tool operations are attribute value relationships for different BTs. One BT attribute — the type of BT — is especially useful in limiting the choices of applicable tools; it doesn't make sense to invoke the C compiler on a BT containing information that is of the "natural language text" type.

Software-life-cycle processes. Ishys models the software life cycle, using the following concepts:

• **Metatasks** — Tasks defining the nature, configuration, and possible orderings of product tasks;
• **Product tasks** — Tasks modifying the informational content of at least one node in the software hypertext;
• **Actions** — Actions needing to be performed to fulfill task commitments; and
• **Primitive actions** — Actions denoting commands performed with Ishys by agents.

Figure 6 depicts the relationship of agent concepts, their tasks, and software hypertext. The following sections elaborate this categorization.

Metatasks. An agent assuming the role of "manager" performs all metatasks. We have identified the following

metatask types (indicating in parentheses the responsible agent category):[6]

• **Planning(PM)** — Detailing the tasks that need to be performed in the software process;
• **Organizing(PM)** — Allocating resources to agents;
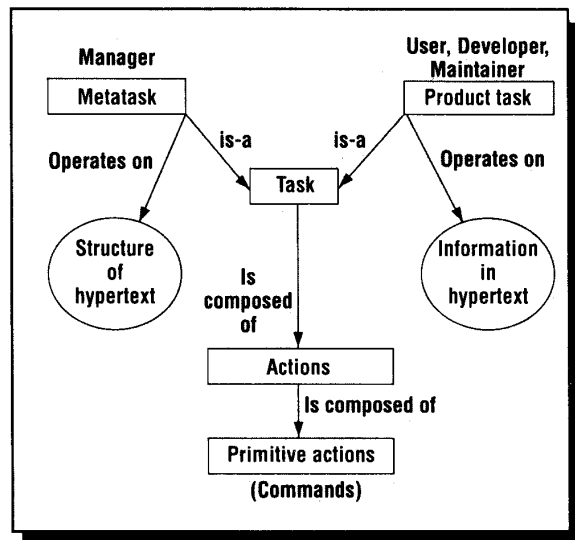• **Staffing(PM)** — Assigning agents to tasks;

Figure 6. Software life cycle products, agents, and tasks.

• **Directing(PM)** — Helping agents in terms of what decisions to make in situations involving uncertain or incomplete information;

• **Coordinating(PM)** — Getting groups of people to work collaboratively;

• **Scheduling(PM)**—Assigning time constraints to tasks;

• **Validating(QAM)** — Confirming that the running system meets user requirements; and

• **Verifying(QAM)** — Confirming the consistency, completeness, and integrity of evolving software product descriptions.

When creating Ishys, we intended to transfer the burden of some of these tasks from people to the system. Hence, Ishys can play the role of a PM by coordinating the tasks of various other agents, based on its knowledge of the tasks they are performing. It can also help human PMs by keeping current information about available project resources, the allocation of resources and tasks to agents, and relationships between various agents and their tasks.

Product tasks. Agents assuming the role of either a Developer, a Maintainer, or a User carry out product-related tasks. Users pose requirements for the system and use the developed system. Maintainers change the system based on emerging requirements of Users. Developers build the system, using requirements posed by Users and guidelines suggested by PMs. Several task types can be performed in this regard, as follows:

User tasks

• — **Requirements definition (Client):** Define the application domain of the target software system and describe how the system fits in the domain.

• — **System use (End user):** Use the system to support some functionality of the end user.

• — **System bug discoveries and reporting (End user):** Use the system, encounter anomalous functionalities, and report errors in the form of bug reports.

• — **Requesting system enhancements (End user):** Realize functionalities that could be added to the running system, and send each enhancement requirement to the appropriate agent.

Developer tasks

• — **Requirements analysis (Analyst):** Verify that user requirements are complete, consistent, and valid.

• — **Functional specifications (Specifier):** Describe the behavior of the target software system's computational objects.

• — **Architectural configuration design (Designer):** Describe the system's structure.

• — **Detailed component design (Designer):** Describe the data structures and algorithms to be processed by individual system modules.

• — **Implementation (Implementer):** Write the source code for system modules in some programming language.

• — **System integration (Integrator):** Package system modules as a whole, possibly combining the result with other systems.

• — **Testing (Tester):** Test the system and its modules against various test case runs.

• — **Usage description (Technical writer):** Write the system's user manual.

• — **System delivery (Integrator):** Deliver the system to customer sites.

• — **Maintenance description (Implementer):** Write the system's maintenance manual.

Maintainer tasks

• — **Fixing bugs in the system:** Analyze bug reports and fix the system and its documents to remove bugs.

• — **Creating system revisions:** Maintain the source code and associated system documents in some revision management system.

• — **Enhancing the system:** Add functionalities to the system for which the system may not originally have been designed and built.

• — **Creating new versions of the system:** Combine various system changes into a new version of the system.

The taxonomy of task types defined for metatasks and product tasks is not meant to be unique or exhaustive. Instead, we support a redefinable taxonomy, thereby associating task configurations with whatever life cycle production model that agents may follow. Several interesting relationships emerge from such a breakup of tasks.

For example, we can organize a software-product-task sequence to follow the conventional "waterfall life cycle" (see Figure 7) or the automation-based paradigm (see Figure 8) suggested by Balzer et al.[8] The differences would depend on task definitions, task (action) sequences, and task (action) relationships.

Boxes in Figures 7 and 8 represent tasks, ellipses represent products, and agents are associated with their appropriate tasks. Arrows from agents to tasks represent the relationship "is-responsible-for"; arrows from products to tasks represent the relationship "uses"; arrows from tasks to products represent the relationship "results-in."

Relationships between tasks (must-precede and must-follow, for example) can be used with a PERT chart to configure tasks in a timing-constraint network. Other
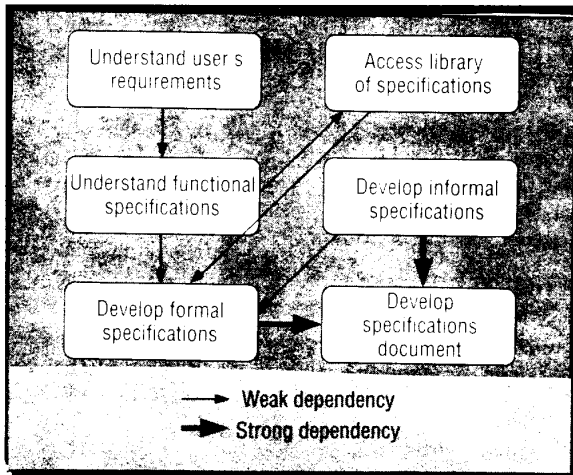
Figure 7. The waterfall life cycle model viewed from product, agent, and task perspectives.

relationships indicate dependencies between tasks. Tasks can be strongly dependent on other tasks, in which case the former task must follow the latter. When tasks are weakly dependent on other tasks, it is preferable (but not necessary) to do the former after the latter is finished. Also, interdependent tasks can require communication or interaction between agents responsible for those tasks.

Results of product tasks are software hypertext nodes. Relationships between hypertext nodes have interesting influences on tasks responsible for creating those relationships and nodes. For example, the relationship "compiled-into" might exist between a source code node and its corresponding object code. Therefore, the task of creating the object code translates into compiling the source code node instead of editing the object code node. The system can utilize the semistructured nature of software hypertext nodes to guide users in creating nodes.

For example, the requirements document can be broken down into nodes describing capabilities that the system must satisfy. Capabilities can be described from a dataflow perspective as a set of input data objects, a set of output data objects, and the name of the transforming capability. To enter system requirements, therefore, the system can give users a Form containing the three appropriate nodes. In addition, based on this knowledge, the system performs consistency checks across various capabilities. For instance, the input set of a capability cannot contain a data object unless it's in the output set of some capability (assuming that the user of the target system is also considered to be a capability in the total dataflow diagram). Naturally, the level of guidance and support users receive depends on the extent

Figure 8. The automation-based life cycle model viewed from product, agent, and task perspectives (adapted from Balzer, Cheatham, and Green[8]).

Figure 9. Actions for the "Develop functional specifications" task.
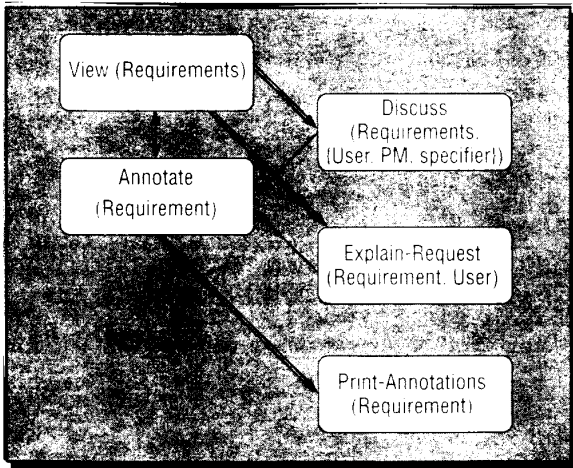

Figure 10. A flow chart of primitive actions for the "Understand user's requirement" action.

to which node contents can be described with a computable grammar; for programming languages, various supporting features (including syntax-directed editing, compiling, debugging, and linking) can be provided. In the case of user manuals, on the other hand, facilities for navigation, search, and completeness (structure) checking can be provided.

Actions. Actions are obtained from task definitions by detailing steps that the agent must complete to effectively carry out the task. As an example, consider the Develop-functional-specifications task — in which an agent plays the role of a Specifier — and which can be broken down into several different actions:

• **Understand the user's requirement;**
• **Understand the system's functional specifications;**

• **Develop the system's informal specifications;**
• **Access the library of exemplar specifications;**
• **Develop the system's formal specifications;** and
• **Develop a specifications document.**

Specifier agents can carry out each action. Figure 9 shows dependencies between actions. We can relate these actions to the BTs defined for the Functional Specifications Form (Figure 4) as follows:

• **"Understand user's requirements" leads to the informal narrative specification BT 2.0;**
• **"Understand functional specifications" and "Develop informal specifications" of the system lead to the narrative specification BT 2.1;**
• **"Access library of (exemplar) specifications" and "Develop formal specifications" of the system lead to BTs 2.2 through 3.0;** and
• **"Develop specifications document" for the system leads to the Functional Specifications Form, with all its BTs integrated.**

Therefore, DIF BTs and Forms are eminently suitable for capturing and integrating the results of multiple actions and tasks. DIF deals only with action and task results, however, and not with how-actions and tasks are performed. Hence, DIF supports actions and tasks at a coarse granularity level, ignoring detailed actions that ultimately make up tasks. In contrast, we focus our attention in Ishys on (1) developing nonprocedural and nondeterministic task definitions in terms of micro-actions composing them, and (2) representing these micro-actions as finely as necessary for automated support. To see how this could be achieved, consider the "Understand user's requirements" action, which is part of the "Develop functional specifications" task. It can be broken into finer-grained actions as follows:

• **View the requirements document;**
• **Create notes regarding the key points of requirements;**
• **Clarify unclear points by communicating with the user;**
• **Discuss (with fellow developers, PMs, or users) points that remain unclear;** and
• **Organize notes about the understanding.**

Primitive actions. Consider hypertext to consist of objects and relationships between objects. Objects are nodes in the hypertext graph, and relationships are edges (in DIF, BTs and links between BTs). From the preceding example, we can identify the following types of primitive action:

• **View(Requirement)** — "Look at" an information object containing a user's requirement;

U-M-I

- **Annotate(Requirement)** — Attach notes of "understanding" to an object X;
- **Explain-Request(Requirement, U)** — Request an explanation of a requirement from a user agent (U);
- **Discuss(Requirement,$\{A_1, A_2, \ldots A_n\}$)** — Discuss a requirement with other agents of the process; and
- **Print-Annotations(Requirements)** — Organize and print annotations attached to requirements.

As Figure 10 shows, the "Understand user's requirements" action can be understood in terms of these primitive actions. Consider Figure 10's diagram as a nondeterministic flow chart. The double-lined arrows show the "elaboration" relation, which means that the action ordering at the arrow's head can be considered as the elaboration of the action at the arrow's tail.

Primitive actions developed in this manner can be generalized by replacing Requirement with an arbitrary object X. For example, we can have an action Understand(X) — composed of View(X), Discuss(X,α), Explain-Request(X,β), Annotate(X), and Print-Annotations(X)—where α is a set of agents, and β is the agent responsible for the creation of X. Using primitive actions, we can readily enhance our DIF approach of combining hypertext and software-engineering-environment features to support various needed capabilities (which we will describe).

Ishys applications

As we have shown, Ishys supports hypertext-based software engineering environment capabilities. In addition, Ishys can help to (1) establish the context of hypertext nodes, (2) accommodate social interactions as work interactions, (3) provide the automatic selection of options for software tools, (4) provide a set of semantically rich commands, (5) support task coordination and different forms of interaction, and (6) provide a basis for experimenting with different factors influencing and influenced by the software life cycle.

Establishing context. The breakup of tasks into actions, and of actions into primitive actions, results in the establishment of the hypertext context. For example, the "Understand user's requirements" action establishes the context in which one of the five primitive actions can be taking place. Therefore, Ishys can automatically prune objects and relationships not pertinent to one of these actions. Since the "Understand user's requirements" action is carried out over time, this can reduce the information overload on agents, as agents need not worry about objects and relationships irrelevant to their immediate action.

In current practice, context establishment is done manually by hypertext users and can lead to much semantic complexity— including getting lost in information space. A context emerging from the task-product-agent perspective minimizes this complexity.

Social interactions and work interactions. One software-life-cycle feature is that the author of information is usually the best source for clarifying concepts about that information. Often, locating the right source becomes a nontrivial task. Ishys can support this endeavor by maintaining a complete history of who did what. Moreover, Ishys has the potential to accommodate social interactions as work interactions. For example, suppose agent A1 communicates (using the mail system embedded in Ishys) with agent A2. Also, suppose that A1 and A2 need not interact to accomplish their tasks. This means that Ishys can determine that A1 and A2 know each other. The degree of their association is indicated (to a certain extent) by the number of messages exchanged between them over time. Consequently, Ishys knows that, if either A1 or A2 is being sought, the other might be able to help.

Tool options. Most tools allow the input of "switches" to tune tool behavior for the application at hand. Consequently, while compiling a C program on Unix, users can give a "g" option informing the compiler that it should generate symbol table information to be used by the symbolic debugger. We can consider such switches as a means for informing tools about some aspect of the environment in which information processing is taking place. In the above case, users are informing the compiler that the code being compiled is an experimental one and is currently being debugged— information required not only for purposes of the switch, but also useful elsewhere (in project status reports, for instance).

Ishys can store this information generically, and then "automatically" generate appropriate switches for the compiler. As another example, consider the "c" option of the Unix system's C compiler, which informs the compiler that the code in the file is part of a bigger system and that the compiler should not use a loader to load the file— information required at another level (namely, the architectural configuration of the system). Hence, the information need be given to Ishys just once, and can be used at multiple places.

Semantically rich commands. Our analysis leads naturally to definitions of primitive actions that can be converted into semantically rich commands. We can provide commands such as Discuss(X,α), which informs Ishys to start a discussion about object X with the set of agents α.

Coordination tools (as suggested by Winograd[9]) can then be integrated with the hypertext system to "intelligently" support discussions between agents. Similarly, a command to Print-Annotations(X) can instruct the hypertext system to

organize and print annotations attached to an object. Appropriate models of node composition can encode what kind of organization is required.[10]

Task coordination. Dividing tasks into the actions that constitute them informs Ishys about agents and what tasks they are expected to perform. This enables Ishys to perform task coordination, by which it can trigger demons when actions that were supposed to be done are not done, or when performing an action requires that an agent respond. For example, if agent A1 performs the action Explain-Request(Requirement, U1), agent U1 is expected to respond with either an explanation, denial, or alternative suggestion.[9] If U1 does not respond within a certain time, a demon can be fired that tells A1 to abandon the request, complain to a manager, or ask someone else. A set of such communication acts arises during a software process, and each set needs automated support for its coordination.[9,11]

Ishys as an experimental tool. In Ishys, we explicitly represent relationships between different factors of the web of computing. Accordingly, Ishys can serve as a tool for experimenting with relationships between web factors. Ishys provides a means for prescribing relationships between agent actions and for recording the action sequence that took place in a specific setting, thereby providing important feedback and increasing our understanding of how work gets done in the software life cycle.[7]

W e have described the design of Ishys, an intelligent software hypertext system, and discussed novel applications that such a system can potentially support. In designing Ishys, we sought to support the software life cycle from a "web of computing" framework — a framework that necessarily requires the consideration of sociotechnical factors influencing and influenced by the software life cycle. Careful analysis of web factors will lead to better designed and more useful software systems.[2,3,5,6,7,9] Ishys supports functionalities that include influencing work interactions based on social interactions, and determining tools and their options based on project status information. Implementation of required enhancements to DIF, our current software hypertext system, has been completed using Prolog, C, and X Windows.

We are continuing to build and experiment with a system that will enable the easy modification of automation rules. This is necessary because — as our experience grows in the use of Ishys and in our empirical understanding of relationships between web factors — we expect Ishys' knowledge base to grow and evolve.

Acknowledgments

References

1. F.P. Brooks, Jr., "Essence and Accidents of Software Engineering," *Computer*, Apr. 1987, pp. 10-19.

2. R. Kling and W. Scacchi, "The Web of Computing: Computing Technology as Social Organization," in *Advances in Computers*, Vol. 21, M. Yovits, ed., Academic Press, Troy, Mo., 1982, pp. 3-90.

3. T. Winograd and F. Flores, *Understanding Computers and Cognition: A New Foundation for Design*, Ablex Publishing, Norwood, N.J., 1987.

4. P.K. Garg and W. Scacchi, "A Hypertext System to Manage Software Life Cycle Documents," scheduled to appear in *IEEE Software*, Jan. 1990.

5. W. Scacchi, "The USC System Factory Project," *ACM Software Engineering Notes*, Jan. 1989.

6. W. Scacchi, "Managing Software Engineering Projects: A Social Analysis," *IEEE Trans. Software Engineering*, Jan. 1984, pp. 49-59.

7. S. Bendifallah and W. Scacchi, "Understanding Software Maintenance Work: An Empirical Analysis," *IEEE Trans. Software Engineering*, Mar. 1987, pp. 311-323.

8. R. Balzer, T.E. Cheatham, and C. Green, "Software Technology in the 1990's: Using a New Paradigm," *Computer*, Nov. 1983, pp. 39-45.

9. T. Winograd, "A Language/Action Perspective on the Design of Cooperative Work," *Proc. Computer-Supported Cooperative Work Conf.*, ACM, New York, N.Y., 1986.

10. P.K. Garg and W. Scacchi, "The Composition of Hypertext Nodes," *Proc. 12th ONLINE Int'l Conf.*, ONLINE Press, London, UK, Dec. 1988, pp. 63-70.

11. B.I. Kedzierski, *Knowledge-Based Communication and Management Support in a System Development Environment*, doctoral dissertation, University of Southwestern Louisiana, Lafayette, La., Nov. 1983 (also available as Tech. Report KES.U.83.3, Kestrel Institute, Palo Alto, Calif.).

Pankaj Garg received his PhD in computer science in 1989 from the University of Southern California in Los Angeles, and is presently a research staff member at Hewlett-Packard Laboratories in Palo Alto. His research interests are in AI, hypertext systems, and software engineering. An all-university predoctoral merit fellow of USC's graduate school from 1984 through 1987, he received his Bachelor of Technology degree in computer science from the Indian Institute of Technology in Kanpur, India. He is a member of the IEEE Computer Society.

The authors can be reached in care of Walt Scacchi, Computer Science Dept., USC, Los Angeles, CA 90089-0782.

Walt Scacchi received his BA in mathematics and his BS in computer science from California State University/Fullerton in 1974, and his PhD in information and computer science from the University of California/Irvine in 1981. Since then, he has been a computer science faculty member at USC. In 1981, he created and has since directed USC's System Factory Project — the only software factory research project in a US university. His research interests include very large scale software engineering, knowledge-based systems supporting the software process, and organizational analysis of system development projects.

An active researcher for more than 50 research publications — and a consulting and visiting scientist with organizations including AT&T Bell Laboratories, MCC, and Carnegie Mellon University's Software Engineering Institute — he is a member of the IEEE Computer Society, IEEE, ACM, AAAI, and the Society for the History of Technology.