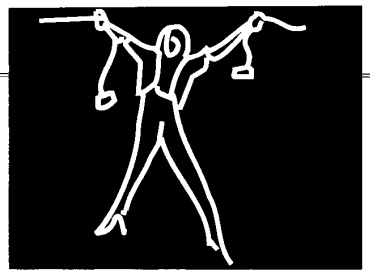


Software Factory Principles, Architecture, and Experiments

CHRISTER FERNSTRÖM, *Cap Gemini Innovation*
KJELL-HÅKAN NÄRFELT, *Telia Research*
LENNART OHLSSON, *Utilia Consult*

◆ *Factory research is progressing from a vision to a reality. The Eureka Software Factory project combines process modeling and an architecture centered on communication to aid integration.*

The software factory concept symbolizes a desired paradigm shift from labor-intensive software production to a more capital-intensive style, in which substantial investments can be made at an acceptable risk level. The software factory represents an evolutionary step on the scale of software-engineering support, a refinement of software-development environments and integrated project-support environments.

The software factory is still more a vision than a reality, but several efforts have been undertaken to realize the concept. One effort is the Eureka Software Factory project, a European effort funded under the Eureka program. ESF is profiled in the box on the facing page.

We foresee a market for software factory parts that can both be configured for specific applications and evolve to take advantage of tomorrow's innovations. Products in this market will range from highly specialized tools to complete environments and will be provided from different vendors, thus requiring vendor-independent integration mechanisms.

To service this market, ESF has defined a communication-centered CASE architecture which, when combined with specific support for describing and animating various software-engineering activities, helps factory builders integrate CASE products.

FACTORY MODEL

The classic factory, where people act as machinery in performing predetermined, repetitive tasks, is neither a desirable nor correct model. In the context of software, the factory analogy can be applied only to

0740-7459/92/0300/0036/$03.00 © IEEE

the *goal* of industrial-style production, not to its *implementation*. The manufacture of software involves little or no traditional production: Every system is unique; only individual parts may repeatedly appear in more than one system.

Most traditional software environments emphasize support for producing code and associated documents. In a software factory, the focus shifts to coordinating information between producers and consumers so that the right person always has the right information at the right time.

The factory analogy applies only to the goal of software production, not its industrial implementation.

Information logistics. There are three perspectives to the logistics of coordinating information:

- At the organizational level, the environment must manage access rights and enforce procedures according to roles and assignments.
- At the team level, the environment must provide change notification and mutability control to manage updates of shared information.
- At the individual engineer level, the environment must reduce information overload and provide focused views of relevant information.

A software-factory environment does not restrict access to information for technical reasons, but for reasons of relevance and possibly policy. The logistics problem is how to provide precisely the information that is currently needed and how to ensure that the information provided is valid and consistent with the assigned user tasks — by ensuring that appropriate validation procedures have been applied, for example.

The software factory has some of the same characteristics as factories that apply computer-integrated manufacturing, especially the continuous focus on synchronizing and integrating independently evolving subprocesses to achieve very broad coverage.

A software factory's scope is the entire enterprise, including all primary and secondary activities related to software production. CIM factories are designed to reduce the isolation of production islands (such as computer-aided design systems, production machines, and ordering systems) while still allowing each subprocess to evolve and be enhanced naturally.

Information accumulation. In the strongly human-oriented software-production process, knowledge and experience are most often collected only informally. Organizations have no memory and carry experience over to new projects by coincidence rather than by design. Manufacturing organizations, on the other hand, measure and analyze production characteristics, which then become important assets in their effort to enhance predictability, quality, and productivity.

An important characteristic of a software factory is the importance it gives to information accumulated from many projects.[1] This information may take many forms, including reusable elements (of code, designs, and documentation), performance measures, development processes, and reports on the effectiveness of applying specific techniques.

To collect and use all this information, an organization must first understand both its semantics and its context. The organization must store the information in semantically rich structures that help consumers understand how to use it effectively. A software factory must have the means to analyze and describe information and its context, store it efficiently (in a knowledge base, for example), retrieve it, and apply it to new situations.

EUREKA SOFTWARE FACTORY

ESF, which began in late 1986, intends to create a market for CASE products that can both be configured for specific applications and evolve.

The ESF consortium comprises 13 partners from five European countries. The companies represent computer manufacturers, research institutions, CASE tool producers, and system developers.

The ESF consortium members are Cap Gemini Innovation, France; EB Tèchnology, Norway; ICL, United Kingdom; Imperial College, United Kingdom; INRIA (Institut National de Recherche en Informatique et en Automatisme), France; Matra Marconi Space, France; Sema Group, France and United Kingdom; Softlab, Germany, Systemhaus GEI GmbH, Germany; TèleSoft, Sweden; the University of Dortmund, Germany; and the University of Durham, UK.

Organized by a management team in Berlin, ESF's activities are distributed across Europe. There are now 15 active subprojects; since 1989, more than 200 man-years per year have been allocated to the ESF project.

By 1991, halfway into the 10-year project, ESF had defined a reference architecture, completed the first implementation of a supporting framework and various tools and tool prototypes, and had undertaken several factory-integration experiments. In the second half of the project, ESF's focus will shift to developing products and introducing its factory concept into organizations. For more information, contact ESF, Hohenzöllerndamm 152, D-1000, Berlin; Internet secretary@esf.de.

REFERENCES

1. Fernström, C., "The Eureka Software Factory: Concepts and Accomplishments," *Proc. Third European Software Eng. Conf.*, A. Lamswerde and A. Fugetta, eds., Springer Verlag, Berlin, 1991.
2. R. Rockwell, "Software Factories: The Industrial Production of 'Mind Stuff,'" *Proc. IFIP Working Group 5 Conf. New Approaches to One-of-a-Kind Production*," Elsevier Press, New York, 1992, to appear.

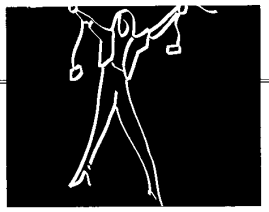


TABLE 1
CASE PRODUCT TYPES

Product	Supported process
Service	Operation
Tool	Task
Tool set	Role
Environment	Production process

EMERGING CASE MARKET

Today's software-production process is so complex and extensive that no CASE vendor can support all the administrative and technical activities of a software enterprise. So CASE developers specialize, providing support for only part of the process. The problem with specialization is that it often leads to products that are isolated and closed — different vendors' products usually do not interoperate correctly. Such a fragmented market prevents the realization of the software factory concept.

The ESF project attempts to minimize fragmentation by creating the conditions necessary for niche vendors to focus on their core business without risking isolation. Each vendor should be able to offer a complementary part of a complete, integrated production environment.

We foresee two types of CASE vendors: *component* vendors, the makers of the factory "equipment," and *factory* vendors, the builders of environments, who select the most suitable equipment, integrate it, and customize it to fit a client's organization and production process.

To build such a market, the technical solution must incorporate customizable, cooperating, heterogeneous components that work together in a distributed environment. Moreover, the software factory concept must cope with evolution: Current technology must be able to coexist with tomorrow's technology.

Product versus process integration. Table 1 lists some CASE products of different granularity.

- A *service* is an atomic operation the user cannot interrupt once it has started, such as cut, paste, and compile. On the process side, a service corresponds to a user function that has been fully automated.
- A *tool* is an integrated set of services, such as editors and project schedulers. On the process side, a tool supports a user task.
- A *tool set* is an integrated set of tools that supports a user role, such as programmer, project manager, and librarian.
- An *environment* is an integrated set of tool sets that supports every role in a software factory.

For products in the same category to interoperate, there must be *interproduct integration.* Services have a strong requirement for interproduct integration, but the requirement decreases as you move down the list of market segments. For example, the ability to cut text is not very interesting if it is not tightly integrated with other editing services. On the other hand, a complete environment need not be tightly integrated with other complete environments.

Tight integration is generally expensive. When a product is tightly integrated, this cost is spread over all the sales of the product. However, environment integration is paid for by each customer, because each environment is customized to an organization. Thus, a strong requirement on the interproduct-integration mechanism is low usage cost.

In contrast to interproduct integration, the requirements for *process integration* — the ability of products to integrate with the organization's processes — are higher for environments than for services. Introducing a service requires very little process knowledge, but introducing a tool set or an environment requires that the roles the product seeks to support match an organization and the methods used by the latter.

The market includes all four product types, so a technical solution to environment construction must cater to two needs:

- The need for a flexible and adequate interproduct integration mechanism that both allows cost-benefit trade-offs for in-

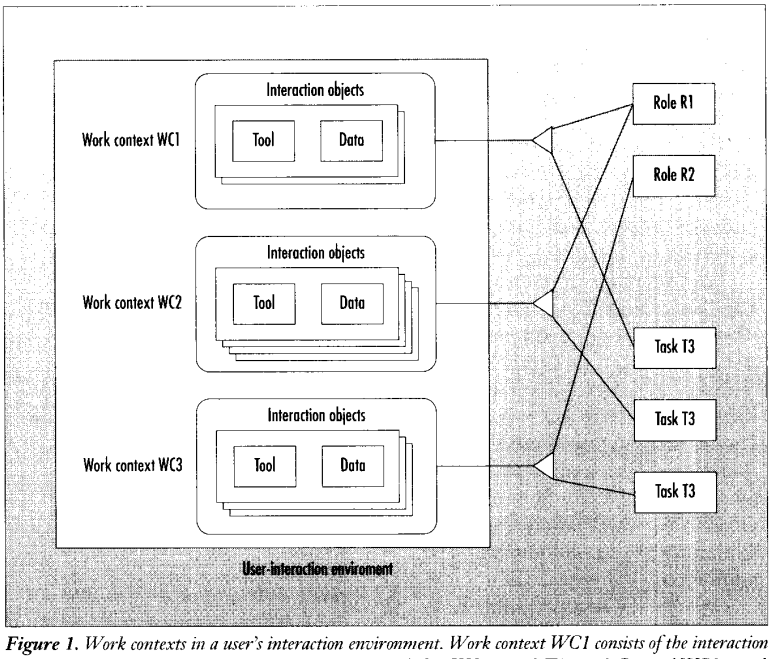


Figure 1. *Work contexts in a user's interaction environment. Work context WC1 consists of the interaction objects related to the user's assigned work on task T1 in role R1; WC2 to task T2 in role R1; and WC3 to task T3 in role R2.*

tegrating products from different vendors and can coexist with the mechanisms used for tight *intraproduct* integration.

♦ The need for process modeling, which lets factory vendors customize their product.

ENVIRONMENT ARCHITECTURE

A CASE environment must provide the right information to the right people at the right time and maintain a consistent view of the system under development, despite the demands placed on it by varied users, data representations, and formalisms.

Process support. Fundamental to realizing a software factory is the formalization of the software-production process. Although it is common to build a process model early in traditional information system development, these models are used mainly as development blueprints.

A process model for a software factory, on the other hand, actively supports users' activities and is used to automate the factory's information logistics. If this process model is expressed in a formal language with executable semantics — as a *process program* — it can be executed while the environment is used.

When the process program and the actual process interact, the process has been *enacted*. Enacting the process brings the environment's functions closer to the needs of the organization and users. In our model, an enacted environment comprises both "hard" software — policy-free building blocks that organizations cannot modify — and "soft" software — process programs that are under the organization's full control.

As Figure 1 illustrates, the user sees process enactment through *work contexts*, a collection of objects with which he interacts and which are specific to his assigned tasks and the roles he plays as he works on them.

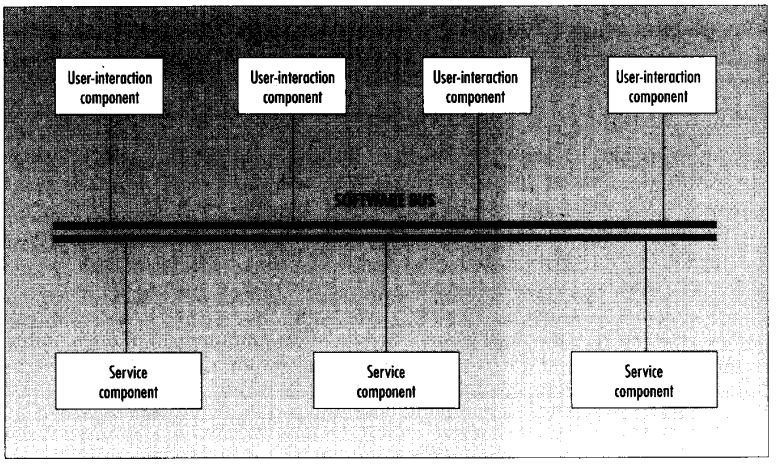


Figure 2. The ESF environment reference architecture. A set of service components and a set of user-interaction components plug into the software bus. Service components represent the concepts supported by the environment; user-interaction components implement the user dialogue. The software bus hides the distribution and heterogeneity of the components plugged into it and puts no restrictions on a component's internal execution environment.

Interaction objects encapsulate information and tools. They may be distinct entities in an object-oriented user interface or they may be independent data and tools. The process program makes work contexts available to a user whenever he is assigned a task and removes the work context when he has completed or delegated the task.

> A CASE environment must maintain a consistent view of the system under development, despite demands placed on it by varied users, data representations, and formalisms.

The user-interaction environment consists of all available work contexts, and so is a view of all the information and functions in a user's support environment. When a user completes a task, the process program may automatically notify other team members or distribute results. An enacted process may also accumulate experience by collecting, classifying, and storing information about the running process automatically.

Environment integration. To integrate the tools in an environment, most recent CASE architectures use a central database. As the box at on p. 40 describes, however, such storage-centered environments have some shortcomings and are not yet widely used.

Figure 2 illustrates ESF's communication-centered architecture, which combines the advantages of highly interactive environments with evolutionary flexibility and the ability to achieve broad coverage.

Environments in ESF are built around a *software bus*, which hides the distribution and heterogeneity of the components plugged into it. The components are islands of tight integration that interoperate via the software bus. The software bus puts no restrictions on a component's internal execution environment. Components may be written in different languages, use private storage models, and independently choose a storage system for managing persistent data. The software bus concerns itself only with how components make their functions available to the rest of the environment. It requires only that they provide a programmatic interface that can be accessed via the protocols the software bus stipulates.

CURRENT CASE ARCHITECTURES

Most recent architectures for software-development environments are based on the use of a central database, or repository, which stores all relevant data.

A common database facilitates the collective use of several tools in many ways. First, its data model imposes a uniform format for all data that is used by more than one tool. Second, its shared state enables the short response times that highly interactive environments require.

Third, a common database schema explicitly expresses the database's intention and how it relates pieces of information. This expression is distinct from how the data is actually stored. Experience has shown that the use of schemas forces developers to give precise information descriptions, thus reducing misinterpretation. Separating semantic data descriptions from storage representation also results in systems that are easy to make small changes to.

Fourth, a common database minimizes control coupling between tools: Because tools operate on the state of the database, rather than the direct output of another tool, they can be ignorant of when the data was produced and what produced it.

Limitations. Despite these apparent advantages, database-centered environments are not (yet) widely used. One reason is that the data models of conventional database-management systems cannot express the rich semantics that CASE applications require (complex integrity constraints and derivation dependencies among entities, for example).

If the schema cannot explicitly state such knowledge, it must be stated in the code. And if several tools share the same data, this code must be included in all of them. For example, in a programming-support environment the compiler, editor, and debugger all have embedded knowledge of the language's syntax and semantics. Such duplication of code reduces tool independence and makes it harder to integrate them.

Another reason for the limited application of database-centered environments is that different tools have very different requirements on issues such as transaction model, query model, aggregation facilities, and information granularity. It is difficult to construct a DBMS that combines a data model of sufficient semantic richness, the generality to cover varying requirements, and the high performance that interactive environments require. This is an area of active research.

So-called language-centered environments, like Interlisp, Smalltalk, and Rational, achieve tight integration by building on special-purpose databases instead of commercial DBMSs. However, because they are specialized, these environments neither fit into a wider context nor cooperate easily with other tools.

Application model. Another, more open, approach is to factor out the parts of the tools that embody the application-specific semantics of the stored data and place them in an *application layer* on top of the data model. Other tools then access the database only through the interface to this layer, the *application model*.

Like the data model, the application model lets schemas be defined. If the application model is object oriented, it lets high-level integrity constraints be implemented as methods, independently of and in combination with an efficient storage model, as expressed by the data model.

A consequence of the distinction between the storage model and the application model when building a tool is something that appears to other tools as a single object may indeed be a collection of storage objects or a computed object with no counterpart in the storage model at all. Because the application model lets you express schemas, building tools this way helps reduce their interdependence.

Components. Inspired by Smalltalk's model-view-controller paradigm, ESF's environments comprise two kinds of components:

- *User-interaction components*, which correspond to Smalltalk's view-controller, let users view and manipulate the data contained in service components. A user-interaction component contains no private application data that survives the session in which it was retrieved or computed. User-interaction components are small, single-user entities designed for a specific activity.
- *Service components*, which correspond to Smalltalk's model, provide functions available to the software bus through programmatic interfaces. A service component encapsulates an internal state that can be accessed and modified by operations available to other components via the software bus. These operations always preserve internal integrity by taking the component from one consistent state to another.

Most of the information managed by a service component is persistent and stored in the component's internal database. ESF explicitly describes a service component's interface in an object-oriented *component model*. As explained in the box at left, such a model lets methods implement high-level integrity constraints independently of the internal data-storage model.

Other components that use the functions provided by service components see the service component as a database with application-specific operations and domain knowledge. Service components are generally large, multiuser subsystems with considerable complexity and correspondingly high price tags.

Process enactment is provided by a *factory process engine*, which implements the runtime support for process programs with a set of service components present in every factory environment. Likewise, service components implement the work-context management functions of the various user-interaction environments and user-interaction components that implement the user interface. Tools generally consist of a user-interaction component acting as a client of one or more service components.

Integration mechanisms. A well-designed service component obeys the rules of strong cohesion and loose external coupling.

Component dependence — the degree to which the inclusion or modification of one component results in the need to include or modify other components — is related to external coupling and to the integration mechanisms' characteristics. A useful guideline to reduce external coupling and achieve strong component cohesion is to map functionality to components in such a way that cycles in client relationships are internal to the components.

ESF is developing two generator tools, SemDraw and Nexus, to automate much of the tedious work in building user interfaces.

The integration mechanisms in the software bus provide three features that largely reduce component dependence:[2]

♦ *Specification-level interoperability.*[3] The software bus provides a common understanding of the data exchanged among components, independent of their actual representation. It supports component integration at build-time with a plug-in mechanism and at runtime with a communication mechanism. Both mechanisms rely on the component models and an environment model, all expressed in the component-description language ESF CDL.

ESF CDL, in turn, comprises three sublanguages: an abstraction-description language, which describes functions independently of representation; a representative-description language, which describes data-interchange formats and control-exchange primitives (for example, synchronous and asynchronous procedure calls or call backs); and a component package-description language, which describes the abstractions a component provides and requires and what representations it uses. The representation parts are used in integration to create and change components' data and control representations as necessary.

♦ *Plug-in mechanism.* The plug-in mechanism incrementally integrates component models with the environment model in a way that lets you introduce components not taken into account earlier. The plug-in mechanism addresses problems like those encountered in developing federated database systems (larger database systems consisting of cooperating database systems that individually developed and maintained.).

As Figure 3 illustrates, the plug-in mechanism uses the environment model to integrate individual components by mapping the components' elements to those of the environment model. The factory builder constructs the environment model from component models in a manner similar to database schema integration. This lets relationships, operations, events, and queries be defined across component boundaries.

The software bus lets you define mappings between the environment model and the component models. This supports the integration process and reduces the coupling between the environment model and components, thereby removing unnecessary dependencies and localizing the effects of changes.

♦ *Communication mechanism.* The communication mechanism provides remote procedure calls and notification-based component interoperation. It provides for synchronous and asynchronous operation, dynamic system reconfiguration, and late binding.

For operations with dynamic binding, the environment model is used for type-driven dispatch. When the software bus receives an operation-object pair, it determines the destination of the operation by looking at the object type that uniquely defines the component on which the operation should be invoked. This may in turn be a more specialized dispatcher. Thus, a client component does not see component boundaries — the environment appears to be an integrated whole.

Component generation. Software development relies on different formalisms and representations, including graphical ones. The choice of formalisms and how they are combined vary from organization to organization, and even from project to project. ESF is developing two generator tools, SemDraw[4] and Nexus,[5] to automate

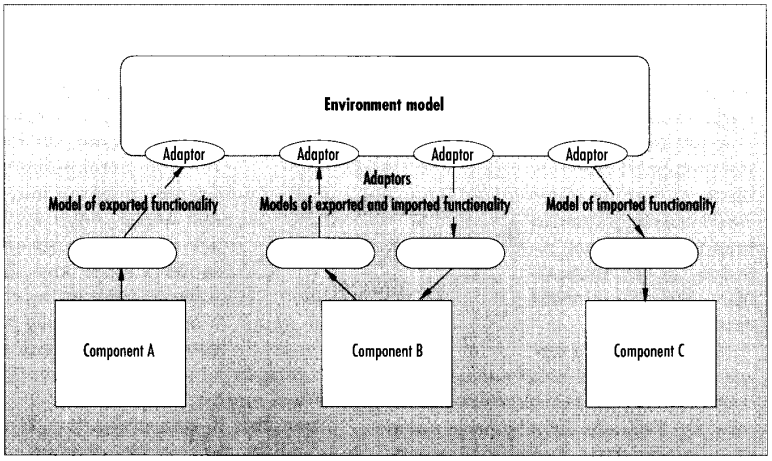
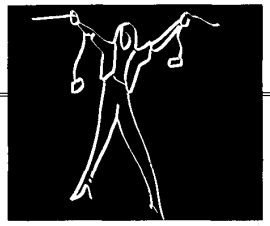


Figure 3. *How components plug into the environment model. Components' exported and imported functionality is mapped to the environment model's elements via adaptors generated by the software bus. The adaptors perform conceptual mappings and transformations of data formats and control exchange primitives. Conceptual mapping involves creating views and correlating one model's objects to another model's objects.*



```
CLASS CompilationUnit IS
    METHOD    compile        [this: CompilationUnit; in: Library]
              RETURNS        [errMsg: Message; linkable: ObjectUnit]
              RAISES         CompilationUnitExceptions
    METHOD-END
    METHOD    check          [this: CompilationUnit; in: Library]
              RETURNS        [errMsg: Message]
              RAISES         CompilationUnitExceptions
    METHOD-END
    METHOD    bind           [this: CompilationUnit; in: Library]
              RETURNS        []
              RAISES         CompilationUnitExceptions
    METHOD-END
    METHOD    nameOf         [this: CompilationUnit]
              RETURNS        [name: string]
              RAISES         CompilationUnitExceptions
    METHOD-END
CLASS-END
```

Figure 4. *Fragment of the Ada service component's model, written in ESF CDL. This fragment describes some of the component's exported functionality using the abstraction-description language of ESF CDL.*

much of the tedious work in building multiformalistic, multiview user interfaces.

SemDraw is a general-purpose drawing tool that lets you define constraints among elements. With SemDraw, you generate specialized editors for different formalisms interactively, extending both the set of predefined elements and the set of predefined constraints.

Different formalisms are often used for different views of the same system: A graphical view might show module decomposition; a textual view might describe semantics. Because views are dependent, it's hard to keep them consistent, especially in environments that support incremental interaction.

A *view server* is a service component that factors out the logical contents of several formalisms and incrementally maintains them by propagating state changes across several components. Nexus is a view-server generator based on attribute-grammar technology, which lets view servers be generated from descriptions of the formalisms used.

FACTORY EXPERIMENTS

An important thread of the ESF project is learning how to integrate large-scale environments. One of our integration experiments has produced a prototype software factory environment for real-time system development; the other, a factory for exploring information logistics.

> The real-time system factory runs on a limited range of systems, but its languages and storage systems are heterogeneous.

Real-time system environment. This environment supports many production activities, including document preparation, project planning, configuration management, design, programming, and quality control.[6] It also supports design and module reuse and the software-production process itself, by mapping the production activities to work contexts, which in turn define the users' tool support.

More than 20 components from eight organizations make up this environment. We developed some of them from scratch, but most were reengineered from commercial tools available from ESF partners and their subcontractors.

The system includes an Ada tool set, developed by TeleSoft, Sema Group, and INRIA (a French national research foundation). This tool set is based on a service component that was reengineered from TeleArcs/TeleGen2, by TeleSoft.[7] This service component provides an object-oriented interface for manipulating Ada objects (such as program libraries, compilation units, program statements, and declarations) according to their semantics and for exploiting their semantic relationships. Figure 4 shows a fragment of ESF CDL that gives the abstract description of operations provided on Ada compilation-unit instances.

A user-interaction component, which we based on the generic language-manipulation system Centaur,[8] provides a highly interactive and adaptable user-interaction environment. This component supports two user roles: system builder and Ada programmer.

For the system builder, the tool set provides facilities for maintaining system baselines and shared library structures. For the programmer, it provides a syntax-oriented Ada editor. Through seamless integration of the user-interaction component and the service component via the software bus, the editor also supports Ada static semantics. The reengineering effort to build the service component, which was implemented in Ada and respected the principles of Ada package encapsulation, was minor. The effort was outweighed by the benefit it added to the supported roles through interoperation with other tool sets like those that support reuse, documentation, and design.

The environment was constructed over 18 months. Development and reengineering was distributed throughout Europe. The integration work, which was supported by prototypes of the software bus and a tool set for process modeling and enactment, was also distributed to a large extent. We usually completed pairwise integration tests within a week.

Although the real-time system factory environment runs on a limited range of systems — Sun Microsystems workstations and PCs — its implementation languages are considerably heterogeneous

— C, C++, Ada, Lisp, Prolog — as are its storage systems — files, Ingres, Oracle, and the Promod object-management system.

Factory for exploring information logistics. Another experimental ESF environment, developed by Cap Gemini Innovation, contains prototypes to support process definition and execution/enactment.[9]

Process designers create and maintain process descriptions with a graphical process editor. They describe the structure of tasks and overall information flow with a graphical notation that is like SADT (Stuctured Analysis and Design Technique), bind tool support and (user) role types to tasks with the usual SADT support links, and define detailed task descriptions and task synchronization with generic, colored Petri nets with preconditions.

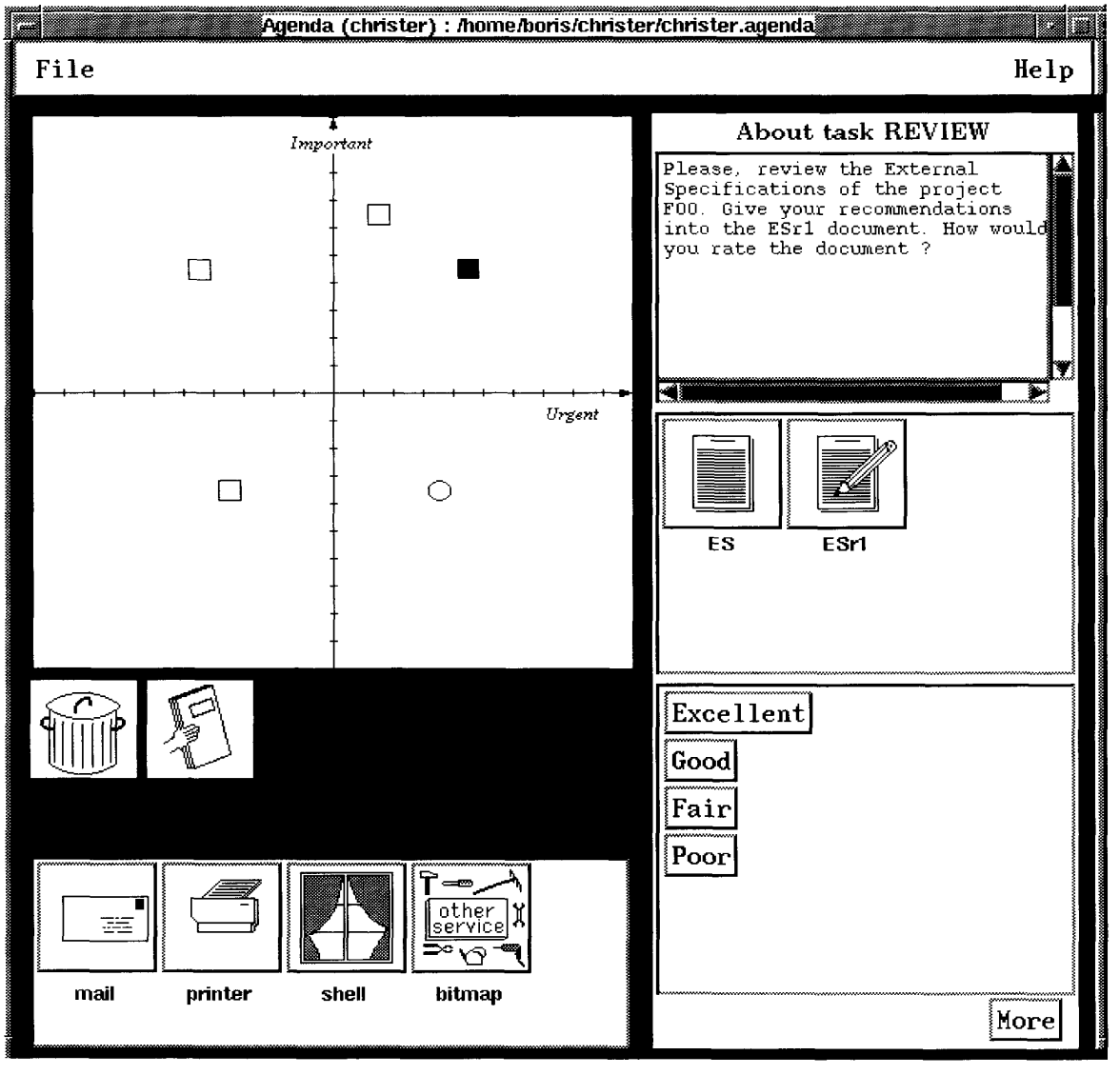


Figure 5. *The main window of the Agenda tool. User tasks are presented in a two-dimensional grid according to their relative importance or urgency. When a task is selected (highlighted), its information is presented in the upper right window; its interaction objects within its work context in the middle right window; and its termination-control buttons in the lower right window. The icons on the left are used for task delegation and nonguided activities.*

To enact a process, the process designer attaches actions, expressed in an action language, to the Petri-net transitions. These actions operate on the factory environment's objects. Typical actions invoke a tool, check out a data object for work, or send a message to a user.

User interaction is based on work contexts that are dispatched to users according to task assignments. The user accesses the environment services with a tool called Agenda. He accesses tools and information through Agenda's object-oriented interface either with process guidance (to perform a planned activity) or without guidance (for example, to read the mail). Either way, access is realized through the activation of a work context that corresponds to a task. As Figure 5 shows, Agenda visualizes tasks many ways: as scrollable lists sorted according to different criteria, as icons, or as points in different kinds of graphs.

Agenda also lets the user create, modify, delete, and delegate tasks and work contexts to other users. The ability to create a work context means the user can group sets of tools and data to support informal tasks, which are managed by Agenda exactly as it manages the tasks it receives via an enacted process. To delegate a task, he simply sends the corresponding work context to the Agenda of the recipient.

So far, we have used this environment to experiment with architectures for process-support environments. In the future we will use it mainly to transfer process technology into organizations.

The fact that the software market has two layers, in which factory vendors provide customized solutions by integrating the products of specialized component vendors, has led the ESF project to focus on support for software processes and component integration.

We have built environments along these lines and early results with prototypes are very encouraging. We are now launching a program to set up software factories in production environments, starting with the European space industry. ♦

ACKNOWLEDGMENTS

The concepts and ideas presented here were developed within ESF, which involves numerous people across Europe, many of whom have actively contributed their ideas to the project. We acknowledge the significant contribution of more than 50 people to the work reported here. Software Bus is a registered trademark of the Eureka Software Factory project.

REFERENCES

1. G. Caldiera and V. Basili, "Identifying and Qualifying Reusable Software Components," *Computer*, Feb. 1991, pp. 61-70.
2. "The ESF Software Bus: An Overview," tech. report, Eureka Software Factory, Berlin, 1991.
3. J.C. Wileden et al., "Specification-Level Interoperability," *Comm. ACM*, May 1991, pp. 72-87.
4. M. Beaudoin-Lafon, B. Chabrier, and M. Thiellement, "Graphics in the Avis UIMS," tech. report, Eureka Software Factory, Berlin, 1990.
5. X. Ceugniet and V. Lextrait, "Integrate Software Development Environments Through Interactive Semantic Views: The View Server Approach," Research Report 90-17, University of Nice, Sophia Antipolis, France, 1990.
6. C. Fernström, "An ESF Pilot Factory for Real-Time Software," *Int'l Conf. Software Eng. Environments*, Ellis Horwood, Chichester, UK, 1991, pp. 305-316.
7. D. Schefström, "Programming-in-the-Large with the System-Oriented Editor," *Proc. Ada-Europe Int'l Conf.*, Cambridge Press, Cambridge, UK, 1988.
8. P. Borras et al., "Centaur: The System," *SIGPlan Notices*, Feb. 1989, pp. 14-24.
9. C. Fernström and L. Ohlsson, "Integration Needs in Process-Enacted Environments," *Proc. Int'l Conf. Software Process*, IEEE CS Press, Los Alamitos, Calif., 1991, pp. 142-158.

Christer Fernström is a chief engineer at Cap Gemini Innovation's research center in Grenoble, France, where he is in charge of research on software process support. He has served as technical director for the Eureka Software Factory project and is currently directing its advanced technology program. His research interests include software systems architecture, development environments, and support for process modeling and enactment.

Fernström received an MS in electrical engineering and a PhD in computer engineering from Lund University in Sweden. He is a member of the ACM and IEEE Computer Society.

Kjell-Håkan Närfelt is manager of a research group at Telia Research, Sweden, the research and development company of the Swedish Telecom Administration. When employed by TeleSoft, he was responsible for the company's activities in the ESF project. His research interests are software-development environments and methodologies and telecommunication-network architectures.

Närfelt received a PhD in computer science from the University of Luleå, Sweden.

Lennart Ohlsson is an independent software consultant. His areas of interest include object-oriented techniques and their implications for software-engineering management, nontechnical aspects of conceptual modeling, and methods for technology transfer from academic research to industrial application.

Ohlsson received an MS in electrical engineering and a PhD in computer engineering from Lund University. He is a member of the IEEE and IEEE Computer Society.

Address questions about this article to Fernström, Cap Gemini Innovation, 7 Chemin du Vieux Chêne - ZIRST, 38 240 Meylan, France; Internet christer@capsogeti.fr.