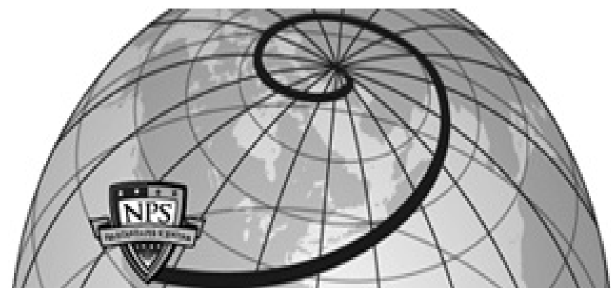
Calhoun
Institutional Archive of the Naval Postgraduate School



Calhoun: The NPS Institutional Archive

DSpace Repository

Faculty and Researchers Faculty and Researchers' Publications

1988

A Computer-Aided Prototyping System

Luqi; Ketabchi, Mohammad

IEEE

http://hdl.handle.net/10945/43616

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

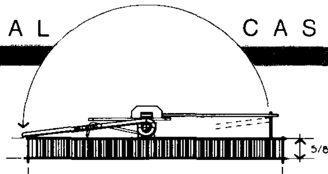
Downloaded from NPS Archive: Calhoun

Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

http://www.nps.edu/library

REAL CASE



A Computer-Aided Prototyping System

Luqi, *Naval Postgraduate School*
Mohammad Ketabchi, *Santa Clara University*

Computer-aided prototyping shows promise. One system under development frees designers from implementation details by executing specifications via reusable components.

A significant improvement in software technology is needed to improve programming productivity and software reliability.[1] Computer-aided, rapid prototyping via specification and reusable components is a promising approach that makes this improvement possible. In this approach, the traditional software life cycle is replaced by a life cycle with two phases: rapid prototyping and automatic program generation.[2]

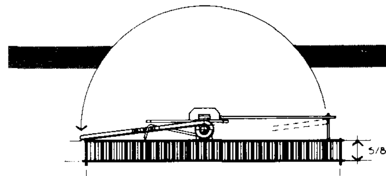
Completely automatic generation of programs from very high-level specifications is not practical today, but automatic generation of prototypes is feasible. Current manual prototyping methods require too much time and effort, but a computer-aided prototyping system would reduce the cost of prototyping and improve the efficiency of the process. However, before such a system can be developed, methods for specifying, selecting, retrieving, and composing reusable components into a prototype that meets a set of requirements must be addressed.

Our approach to rapid prototyping uses a specification language (the prototype-system description language PSDL) integrated with a set of software tools, including an execution support system, a rewrite system, a syntax-directed editor with graphics capabilities, a software base, a design database, and a design-management system. The prototyping language lets the designer use dataflow diagrams with nonprocedural control constraints as part of the specification of a hierarchically structured prototype. The resulting description is free from programming-level details, in contrast to prototypes constructed with a programming language.

The underlying computational model unifies dataflow and control flow, providing a vehicle for developing top-down decompositions. Such decompositions let large prototypes be executed with practical computation times, in contrast to prototyping by simulating specifications via logic programming without providing a system architecture.

The prototype is executed with the aid of reusable components drawn from a software base. The prototyping language is an integral part of the design-management system because specifications are used to organize and retrieve reusable components in the software base.

0740-7459/88/0300/0066/$01.00 ©1988 IEEE



A rewrite system makes retrievals more effective by reducing syntactic variations in equivalent retrieval requests.

The retrieval mechanism does limited bottom-up design to compose requested components without requiring the designer to be aware of all the modules in a large software base. Retrievals based on formal specifications can be made more selective than those based on keywords, reducing the number of inappropriately retrieved components examined by the designer.

Specifications are better for retrieval than implementations because the properties of implementations are too difficult to recognize mechanically. It is not feasible to automatically choose routines from a conventional program library without special annotations.

Figure 1 illustrates the major steps in computer-aided prototyping. The designer begins the process by entering the specifications of the intended software component. A rewrite subsystem maps the specification into an internal abstract form that is used by a design-management system to search for the software component. If it finds a unique software component that meets the specification, it retrieves the component; if it finds several software components that meet the specification, the designer must choose one. Otherwise, the specification cannot be met by an existing component and the designer should decompose the specification into simpler specifications by using the system's prototyping language.

When a specification is decomposed into a network of simpler components, the required interconnections are recorded in the design database with a dataflow diagram, which is part of the syntax of the prototyping language and serves as design documentation. After the designer decomposes the specification into simpler specifications, the entire process is applied to those specifications.

Information about how the system can compose components that meet the simpler specifications into a component that meets the requested specification is preserved in the design database when the designer decomposes the specification. The designer uses that information to compose the missing implementation of the requested specification after the implementations of simpler components become available. If the designer cannot decompose the specification, a new component must be hand-coded and stored in the component base.

The computer-aided prototyping system reduces the designer's efforts by automating time-consuming tasks in conventional prototyping, such as turning specifications into prototypes, modifying prototypes,

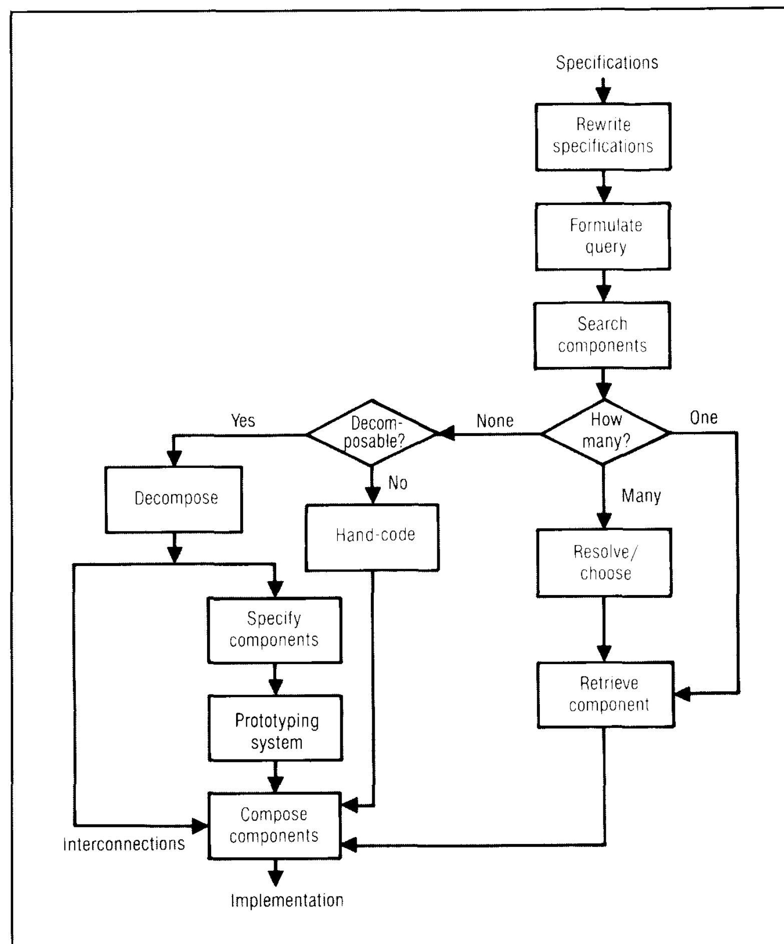


Figure 1. Prototype development using the computer-aided prototyping system.

and searching for available reusable components.

Software tools are needed to support this automated prototyping method. The major parts of such a system are

- the Prototype System Description Language,[3]
- user interfaces to speed up design entry and prevent syntax errors,
- an execution-support system to demonstrate and measure prototype behavior and to perform static analyses of the prototype design,
- a design-management system to manage reusable software components and design data,
- a software base to store reusable components, and
- a design database to store the prototype design.

Yeh and colleagues[2] have proposed an initial framework for a rapid-prototyping environment based on reusability. We further developed a prototyping method, an executable prototyping language, its execution-support system, and better automated methods for component organization and retrieval using normalized specifications. Figure 2 shows the architecture of a prototyping system that supports the process shown in Figure 1.

Language and method

A good language for expressing design thoughts in terms of a precise model is important for rapid prototyping. It is impossible to do a good design without a language designed especially for this purpose. A powerful, easy-to-use, and portable prototype-description language is a critical part of a computer-aided prototyping system. Such a language is needed before the tools in the system can be built.

PSDL was designed together with the prototyping method[4] to ensure the most efficient use of the language. It serves as an executable prototyping language at a specification or design level and has special features for real-time system design.

PSDL provides two kinds of basic building blocks for prototypes: data types and operators. These constructs are sufficient to specify a prototype's design and structure. Software systems are modeled as networks of operators communicating via data streams. The networks are represented as dataflow diagrams with a bubble for each operator and an arrow for each data stream. The data streams can carry data values of an abstract data type as well as tokens representing exception conditions. PSDL provides graphical notation for dataflow diagrams enhanced with nonprocedural control and timing constraints. A formal syntax describes these constraints and other attributes for specifying a prototype.

Each operator is atomic or composite. Atomic operators are realized by retrieving an implementation from a software base.[5,6] Composite operators are realized by decomposing them into networks of more primitive operators represented as enhanced dataflow diagrams. Both atomic and composite operators are used as components of prototypes.

Good modularity is important for increasing productivity because it significantly reduces the debugging effort for producing a correct, executable system. It also influences the system's understandability, reliability, and maintainability, which are especially important in rapid prototyping.

The PSDL computational model is based on dataflow under semantically unified control and timing constraints. It prevents hidden interactions between system components to encourage designs with good module independence since dataflow provides simple and clear interfaces

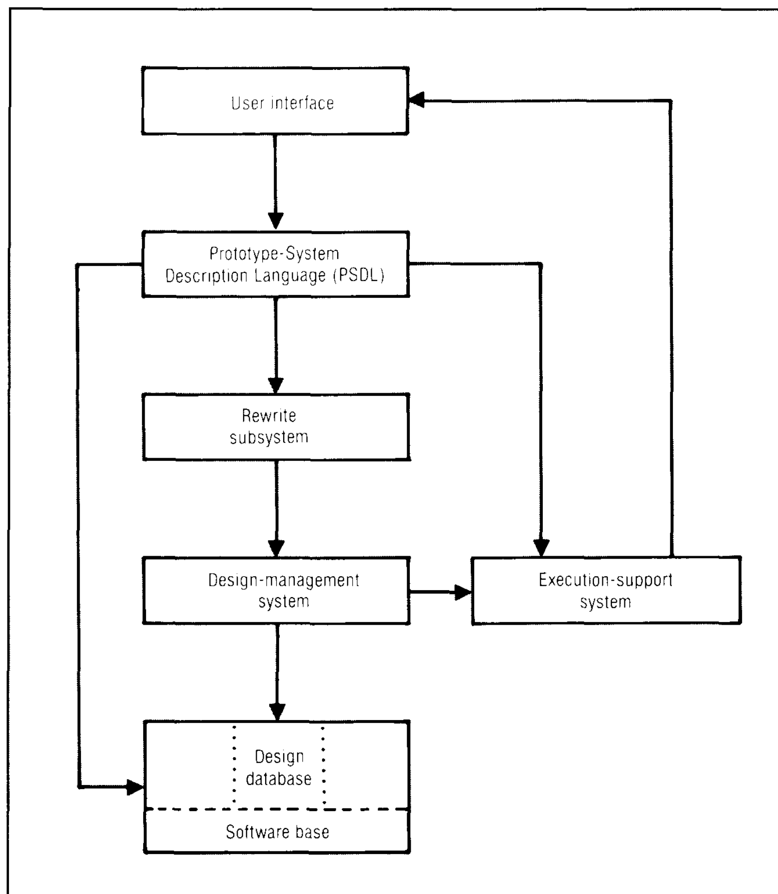


Figure 2. Prototyping system architecture.

between operators, since all data inside operators is local, and since operators with internal states cannot be implicitly shared. The worst coupling problems caused by external references in common control-flow structures are eliminated completely.

The nonprocedural control constraints are easy to use because their meaning does not depend on the order in which they appear. Control constraints make execution more efficient and provide more flexible input and output facilities for triggering operators and selectively generating data values than conventional dataflow. We use a clear and powerful modularization model for building and describing the prototype.

Figure 3 shows an enhanced dataflow diagram with operators *A*, *B*, and *C* and data streams *a, b, c, d, e,* and *f*. The maximum execution times of the operators are 10 ms for *A*, 20 ms for *B*, and 10 ms for *C*. The control constraint on *A* says the output *d* is produced if *a* equals *c* and is suppressed otherwise. The control constraint on *B* says the operator is triggered whenever new data arrives on *b* or *d*. If a new data value arrives on stream *b*, the operator fires using the new value of *b* and the most recent data value it has read from *d*.

We combine control constraints with the dataflow model to achieve the best modularity with sufficient control information, and use dataflow to simplify the interactions among modules, eliminating direct external references and communication caused by side effects.

The language and its associated prototyping method[7] lead to PSDL prototypes with a highly cohesive structure and few coupling problems because they support the model and combine it with a powerful set of data and control abstractions to make it easy to describe systems at a high level. This structure is suitable for multiple modifications at a specification level during the prototyping iterations of the new life cycle.

The PSDL prototyping method provides a hierarchical decomposition strategy for filling in more design details at any level of the prototype design. It helps the designer concentrate on the critical subsystems that must be refined. The prototyping method uses stepwise refinement to selectively refine and decompose critical components. These refinements and decompositions are kept in the design database. Each higher level component is described in terms of lower level components and the relations among them. The decomposition of each composite component is a realization of the system at a lower level of detail.

The prototype design is based on abstract functions, abstract data, and abstract control. This high-level view emphasizes the overall configuration at each level without bogging down in programming-level details. The designer refines the design by decomposing abstract functions and data types into lower level ones. Functional, data, and control abstractions can be used to hide lower level details.

Rewrite subsystem

We based our approach to component specifications on term rewriting, which reduces the variations in the representation of software specifications. We call this approach normalizing, which is mapping semantically equivalent specifications to a common form that is used by the design-management system to search for components. Normalized components are easier to retrieve because there are fewer keys to search for in the software base and because the information is stored in a standard form. The designers can choose from several specifications, but the information-retrieval system is not burdened with handling all these variations because the designers' specifications are automatically normalized before storage.

Because there can be many syntactic forms for the same semantic description, reduction to a normal form is a more practical approach than trying to generate all variations of a description and searching the software base for each variation. Table 1 shows an example of an informal term-rewriting system.

The rewrite rule defined by such a table simply replaces all occurrences of the aliases by the associated basic terms. The sentence "Fetch the order from the transaction file and modify the inventory" would be rewritten to "Read the order from the transaction file and update the inventory."

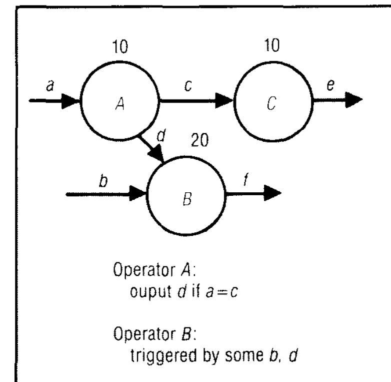


Figure 3. Enhanced dataflow diagram with control constraints.

The rewrite subsystem translates equivalent specifications into normalized specifications (see Figure 4) that will be used by the design-management system to find and retrieve the required components from the software base. Two kinds of normalization techniques, for formal and informal specifications,[8] store the normalized specifications with the components in the software base.

Design manager

The design-management system is responsible for organizing, retrieving, and instantiating reusable components from the software base and for managing the versions, refinements, and alternatives of prototypes. A design-management system must efficiently select and retrieve the relevant components from a software base because, for computer-aided prototyping to be practical, the retrieval must take less effort than constructing the components.

A design-management system is essentially a database-management system that can efficiently manage long transactions, data describing complex objects (such as software components), the iterative and tentative nature of the design process that leads to versions, refinements, and alternatives of the design objects, and concurrent design operations in a distributed computing environment. It also provides special-purpose operations to compose components, browse the software base, and manipulate the normalized specifi-

Table 1.
Sample rewrite-subsystem rule table.

Term	Aliases
update	change, modify, refresh, replace, substitute
read	fetch, obtain, input, get, retrieve

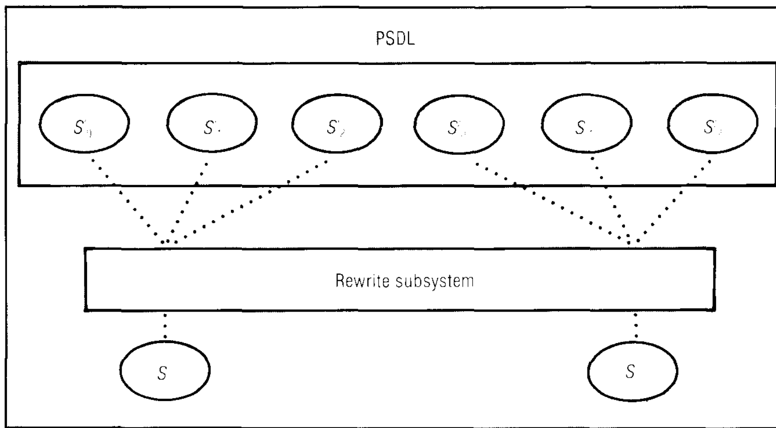


Figure 4. Normalizing specifications

cations.

We have compared conventional database-management systems with the requirements of computer-aided design applications. Because conventional database-management systems do not meet the data-management requirements of CAD applications, we are developing an object-oriented database-management system based on our object-oriented data model[9] for the design-management systems. Our data model meets the requirements of CAD applications with considerable simplicity and economy of concepts. We will tailor the model to meet the requirements of prototyping systems, but we do not exclude the use of a commercial database-management systems to experiment with and study the features of a prototyping system.

Software base

Reusable components must be self-contained and portable. The software base must be easily extensible to allow the evolutionary growth of the available components. You must be able to browse, select, and retrieve components from the software base efficiently. To achieve these goals, we are developing a highly structured software base. Three major structural foundations of the software base are generalization by category, specification approximation, and component composition.

Components have certain properties, called categorical properties, that are used to categorize them. Figure 5 shows categorical properties "implementation languages" and "system environments" with some of their subcategories. Generalizing components according to their properties imposes a lattice structure on the set of components.[10] Figure 6 shows the generalization lattice of components based on the categorical properties "implementation languages" and "system environments."

The lattice structure of components allows efficient browsing of the software base and supports efficient selection and retrieval of the components by partitioning the set of components into meaningful subsets. For example, if an Ada component for the Digital Equipment Corp. VAX/VMS environment is needed, only those components that belong to VMS_Ada node of the lattice are of interest. If an Ada component is needed and the system environment is irrelevant, only those components that belong to Ada node of the lattice must be examined.

In general, there is more than one component in each subset generated by generalization per category. The specification of the desired component is used to select a unique component in a subset.

The software base contains a large set of normalized specifications corresponding to unique implementations. These specifications are called singleton specifications (from the card-playing term for a card that is the only one of its suit held in a hand). A pair containing a singleton specification and its implementation is a singleton component. A normalized specification requested by the designer may not be singleton but an approximation of some singleton specifications. A specification S_i is an approximation of a specification S_j if S_j implies S_i. In this case, S_j is a refinement of S_i.

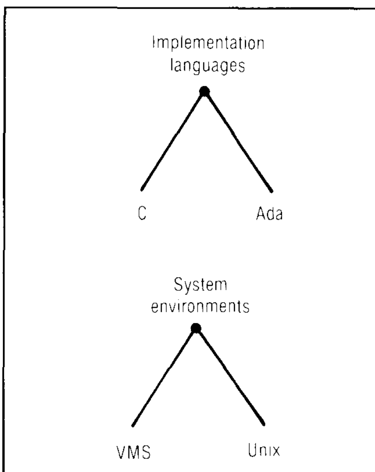


Figure 5. Categorical subproperties with subcategories.

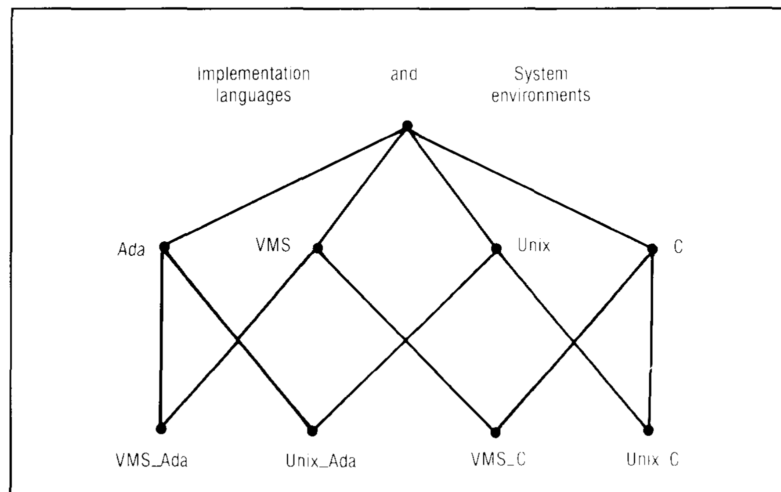


Figure 6. Generalization lattice of categorical properties.

The design-management system lets you derive the best approximation of a set of singleton specifications. The set of singleton specifications and their approximations has a lattice structure, and the explicitly stored specifications are the basis of this lattice. Other specifications can be derived from singleton specifications. Figure 7 illustrates a software base with four singleton specifications (S_s) and three approximate specifications (S_a). If the specification of a requested component matches a singleton specification, the system retrieves the implementation. If it matches one of the approximate specifications, the designer can select any refinement of that specification. Otherwise, a new singleton component is hand-coded and inserted into the base.

The design-management system controls the insertion of components into the base and updates the approximation lattice after each insertion. Creating a new singleton component becomes necessary only if the specification cannot be decomposed into simpler specifications. When a specification is decomposed, the design-management system builds a composition template. A composition template contains the specifications of the components that are needed to construct the requested component.

Components that meet the composite specifications are called composite components. (Singleton components, by contrast, are atomic.) Composite components are virtual because only the recipe for their construction is stored. However, the design-management system may cache the implementations of composite templates that are used frequently.

Execution support

The PSDL execution-support system contains a translator, static scheduler, and dynamic scheduler. The translator generates code binding together the reusable components extracted from the software base. Its main functions are to implement data streams and control constraints. The static scheduler allocates time slots for operators with real-time constraints. If the allocation succeeds, all operators are guaranteed to meet their deadlines even with worst-case execution times. The dynamic scheduler invokes operators without real-time constraints in the time slots not used by the operators with real-time constraints. The dynamic scheduler also lets the designer control and examine the execution of the prototype.

Our research addresses several key problems in automated prototyping with reusable software. These problems include conceptual design of an integrated prototyping system, prototyping language and methodology, normal-form specification for reusable components, a design-database-management system, and a software base that supports efficient retrieval of components by their specifications.

The computer-aided prototyping system combines a high-level prototyping language, a systematic design method to rapidly construct prototypes, a large software base, and a design-database-management system. We believe the system will sharply reduce the need for requirements changes after implementation has begun, as well as for many requirements changes during the design of a new feature in an evolving system.

Demonstrating the prototypes constructed with the prototyping system will give users feedback early enough in the development cycle so they can extensively adapt the design without wasting a lot of effort. This should lead to products that closely match users' needs.

Our approach uses specifications as an intimate part of the computer-aided implementation, making documentation a natural by-product of development rather than a costly extra task and helping to ensure that the documentation corresponds to what the system actually does. The specification of a prototype written in PSDL provides formal documentation for the system, which is a hierarchically structured design with specifications of all components and the interconnections among them. The language syntax includes dataflow diagrams. Informal English explanations of the decomposition represented in the dataflow diagram can be generated by a paraphraser.

The prototyping system is extensible because it provides facilities for adding new components to its software base. The integrated approach to the maintenance and the management of prototype design data simplifies the adaptation of new tools and techniques. It also provides a knowledge base for expert design and analysis tools.

Acknowledgments

This work was supported in part by the National Science Foundation under grant CCR-8710737. We thank the editors for their extensive comments, which significantly improved this article.

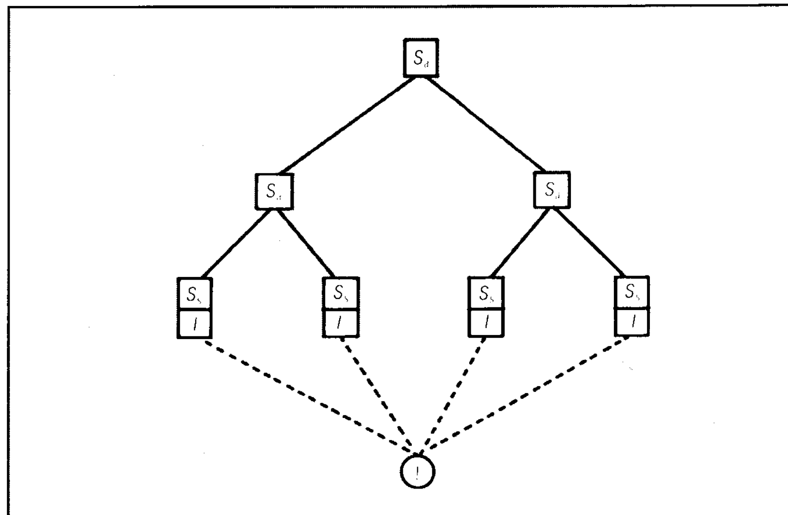


Figure 7. A component base with four singleton components.

References

1. V. Berzins, M. Gray, and D. Naumann, "Abstraction-Based Software Development," *Comm. ACM*, May 1986, pp. 402-415.
2. R.T. Yeh et al., "A Programming Environment Framework Based on Reusability," *Proc. Int'l Conf. Data Engineering*, CS Press, Los Alamitos, Calif., 1984, pp. 277-280.
3. Luqi, V. Berzins, and R. Yeh, "A Prototyping Language for Real-Time Software," Tech. Report 86-4, Computer Science Dept., University of Minnesota, Minneapolis, Minn., 1986; to appear in *IEEE Trans. Software Eng.*, 1988.
4. Luqi, *Rapid Prototyping for Large Software-System Design*, doctoral dissertation, Computer Science Dept., Univ. of Minnesota, Minneapolis, Minn., June 1986.
5. N. Roussopoulos, "Architectural Design of the SBMS," tech. report, Computer Science Dept., Univ. of Maryland, College Park, Md., April 1985.
6. R.T. Yeh, N. Roussopoulos, and B. Chu, "Management of Reusable Software," *Proc. Compcon*, CS Press, Los Alamitos, Calif., 1984, pp. 311-320.
7. Luqi and V. Berzins, "Rapid Prototyping of Real-Time Systems," Tech. Report 87-5, Computer Science Dept., Naval Postgraduate School, Monterey, Calif., 1987; to appear in *IEEE Software*, 1988.
8. Luqi, "Normalized Specifications for Identifying Reusable Software," *Proc. Fall Joint Computer Conf.*, CS Press, Los Alamitos, Calif., 1987, pp. 46-49.
9. M. Ketabchi and V. Berzins, "Modeling and Managing CAD Databases," *Computer*, Feb. 1987, pp. 93-102.
10. M. Ketabchi, V. Berzins, and K. Maly, "Generalization Per Category: Theory and Application," *Proc. Int'l Conf. Information Systems*, ACM, New York, Dec. 1985, pp. 227-237.

Carnegie-Mellon University
Software Engineering Institute

The future is being defined at the Software Engineering Institute

The Software Engineering Institute is building the future of software engineering in the United States, and we need talented, productive people to achieve that goal. We are seeking experienced technical staff at all degree levels who possess strong communication skills.

The SEI needs experienced software engineering practitioners who understand the problems in developing large-scale software systems. We offer a wide variety of opportunities in the software process, Ada development, and support environments for real-time systems including database design.

The Software Engineering Institute serves the public interest by accelerating the transition of software technology into widespread use. It is located in Pittsburgh, the most livable city in the country, which is rapidly becoming one of the nation's leading centers for high technology.

Explore the challenge of tomorrow's technology by building your career with the Software Engineering Institute.

● Send your resume to:

Sally Miller D10
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
(412) 268–5678

The Software Engineering Institute is sponsored by the Department of Defense. Carnegie Mellon University is an equal opportunity/affirmative action employer.
U.S. citizenship or resident alien status required.

Luqi is an assistant professor at the Naval Postgraduate School. She has also worked on software research and development and taught at University of Minnesota. Her interests include software development tools, rapid prototyping, and languages.

Luqi received a BS in computational mathematics from Jilin University, China, and an MS and PhD in computer science from the University of Minnesota.

Mohammad A. Ketabchi is an assistant professor of electrical engineering and computer science at Santa Clara University. His research interests include database support for computer-aided design and software engineering and design and implementation of object-oriented systems.

Ketabchi received a BS in electrical engineering and an MS and PhD in computer science, all from the University of Minnesota.

Address questions to Luqi at Computer Science Dept., Naval Postgraduate School, Monterey, CA 93943.