Technical Report

CMU/SEI-88-TR-22
ESD-TR-88-023

Carnegie-Mellon University
Software Engineering Institute

Perspective on Software Reuse

J. M. Perry
GTE Resident Affiliate
GTE Government Systems Corporation

September 1988

ADA204399

Technical Report
CMU/SEI-88-TR-22
ESD-TR-88-023
September 1988

Perspective on Software Reuse

J. M. Perry

GTE Resident Affiliate
GTE Government Systems Corporation

Application of Reusable Software
Components Project

Approved for public release.
Distribution unlimited.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

SEI Joint Program Office
ESD/XRS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

Karl H. Shingler
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Copyright © 1988 by Carnegie Mellon University.

This document is available through the Defense Technical Information Center. DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145.

Copies of this document are also available through the National Technical Information Services. For information on ordering, please contact NTIS directly: National Technical Information Services, U.S. Department of Commerce, Springfield, VA 22161.

Use of any trademark in this publication is not intended in any way to infringe on the rights of the trademark holder.

Table of Contents

Acknowledgement 0

1. Introduction 1

2. Ideal Problem Solving Capabilities 3

3. Ideal Software Development Capabilities and Software Reuse as a Practical Approximation 7

4. Integration of Development Capabilities 11

5. Generic Model Improvement 15

6. Features Analysis and Initial Formulation of the Generic Development and Application Model 17

7. Conclusion and Recommendation 19

References 21

Acknowledgement

GTE, the SEI, and the Application of Reusable Components Project provided the opportunity to formulate and refine the ideas of this paper.

The report was edited by Linda Hutz Pesante who, both as a technical writer and objective reader, contributed to the expression of these ideas and the completion of the report.

Perspective on Software Reuse

Abstract: This report presents a perspective on software reuse in the context of "ideal" software development capabilities. Software reuse is viewed as a means of achieving—or at least approximating—the ideal capabilities. A generic application and development model is proposed for unifying various types of software reuse. The model can be initially formulated as a project family architecture and produced from a domain features analysis. The approach presented in this report is intended to lead to a reuse strategy and methodology for software development.

1. Introduction

This report focuses on an approach to software reuse which can be expanded to a reuse strategy for software development. We assume a methodology which divides the life cycle into phases, such as requirements through integration and maintenance, where each phase produces intermediate artifacts.

The report views software development as a form of means-ends analysis, a problem-solving method that formulates a problem as a search for a sequence of actions beginning with an initial situation or state and ending with a goal situation or state. At each state, an operator is selected which reduces the difference between the current state and the goal. A solution to a problem is a sequence of operators, the sequence of states (from initial through intermediate to the goal), and the constraints which the sequence satisfies. In this setting, the initial state is the beginning of a software development, that is, the software requirements and system description; a goal is the software system product; and the operators are methods for transforming the artifacts. Software engineering attempts to reduce the search aspects of the process by formulating methodologies which specify the type of the intermediate artifacts and guide the application of operators.

In means-end analysis, analogical problem solving is the process of retrieving a solution to a similar, previously solved problem and transforming it into a solution of the current problem. Using this approach, software reuse can be precisely formulated as analogical development, where a previous development from similar requirements is transformed to a new development satisfying new constraints. An important point is that the selection of operators corresponds to design decisions. This view gives us insight into two reasons systematic software reuse is difficult. First, the complete solution to the previous software development is typically not available; usually, only the artifacts are available, and design decision information is incomplete. Secondly, the constraints of the solution to the current development may differ greatly from the constraints of the previous development.

This view of software reuse as the use of "sequences" of software development "solutions" of previous projects to "solve" a current development underlies many of the ideas of the following discussion.

2. Ideal Problem Solving Capabilities

To put software reuse in perspective, this section describes "ideal" capabilities for problem solving; the next section reformulates them for software development. Finally, reuse is presented as a way of achieving (or, at least, of approximating) these desired "ideal" capabilities. The capability is "ideal" in the sense of being powerful and efficient for certain application domains, and desirable but not yet available for other application domains. The "ideal" capabilities are presented as problem-solving scenarios. In the next section, they are recast as software development scenarios so that they can become the basis for a reuse strategy.

Problem solving, including software development, is a goal-directed process which attempts to produce products (i.e., goal artifacts) from initial artifacts, such as requirements or specifications. The development involves, to varying degrees, both constructive and derivative processes. For example, the products are constructed from more primitive components, and artifacts are transformed from initial phases to final phases. The emphasis in the development will vary between construction and derivation, depending on the initial artifacts available.

The scenarios which illustrate this goal-directed process follow a means-end problem-solving paradigm and use the terms *initial artifacts*, *goal artifacts*, *tasks for deriving the goal artifacts*, and *intermediate artifacts*, rather than the terms *initial state*, *goal state*, *operations*, and *intermediate states*. The following scenarios present several "ideal" capabilities. They involve:

- comparison of artifacts
- transformation and derivation of artifacts
- comparison of tasks
- transformation of tasks

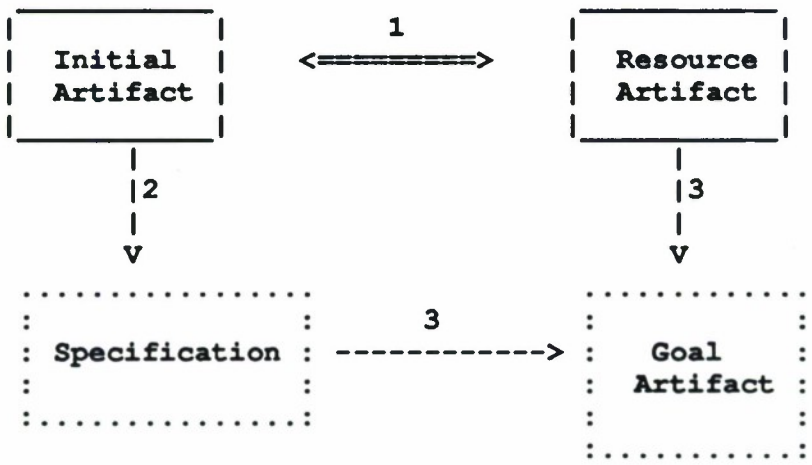
The first scenario posits an initial artifact and a resource artifact, a means of comparing them, a way of using the comparison to go from an initial artifact to a specification of the goal artifact, and a formal way of deriving the goal from the specification and the resource.

The second scenario posits an artifact similar to the goal artifact, a way of comparing the initial artifact with the similar artifact and of comparing constraints of the tasks used for producing the similar artifact with those of the current development, and a means of using the comparison results to transform the previous development to a current development of the goal.

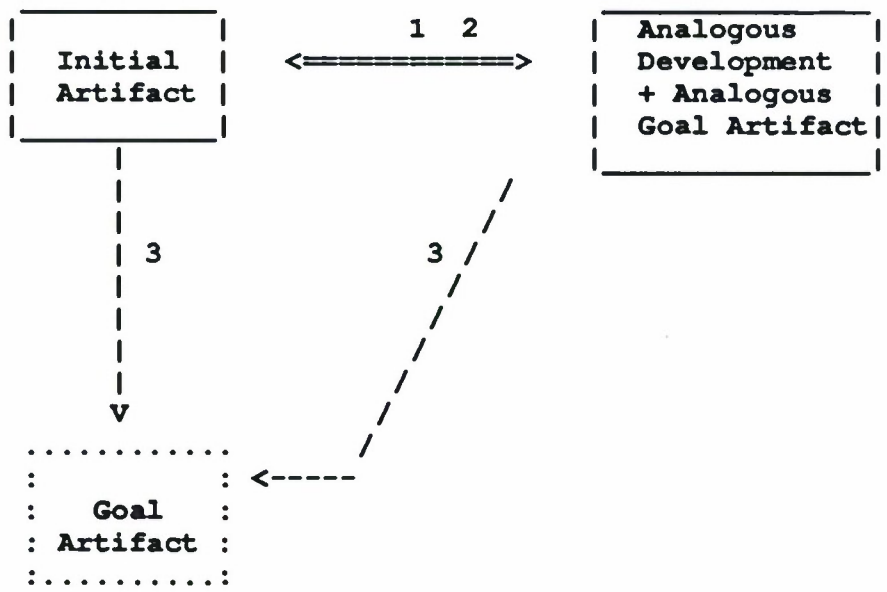
The third scenario posits an initial artifact, a collection of available artifacts of various kinds, a way of comparing the initial artifact with the collection, and a way of using the comparison results to derive the goal from the initial artifact and parts of the collection.

The following three diagrams summarize and help clarify the three scenarios. In the diagrams, the double arrows denote comparisons, the dashed boxes denote starting artifacts (initial, resource, or analogous); dashed arrows denote transformations or derivations to be carried out; and the dotted boxes, goal artifacts. The digits indicate the order in which the comparisons and transformations take place. The initial artifact is, typically, a description of the problem to be solved and the constraints which apply to a solution.

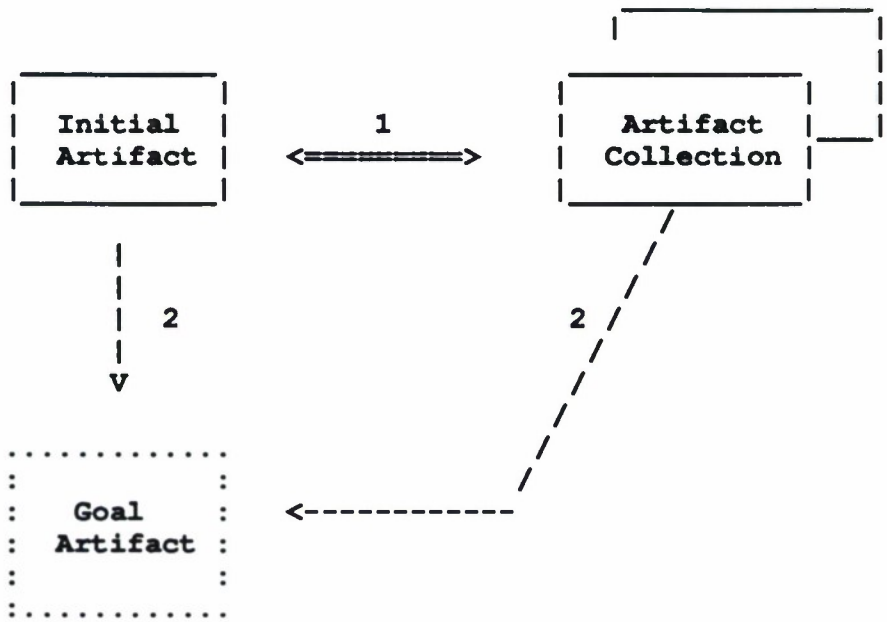
Scenario One



Scenario Two



Scenario Three



3. Ideal Software Development Capabilities and Software Reuse as a Practical Approximation

The three scenarios include a variety of software development approaches that can be applied to software reuse. The first scenario illustrates the capability of using a general application resource and involves requirements in the form of desired operational behavior. For example, a database system can satisfy varied storage and retrieval types of operations. The desired storage and retrieval behavior can be described using database commands which are interpreted by the database system to carry out the desired operational behavior. In this scenario, the initial artifact is the desired operation, the resource artifact is the database system, and the goal is a command set that has the desired operational behavior. The command set is a goal because the commands together with the resource constitute a system that will carry out the desired operations.

The second scenario involves requirements in the form of desired features of a common product. A feature which is a desired characteristic of a system may pertain to either the structure or the operation of the system. Typically, a user feature characterizes the operational capability of the system. For example, a particular VCR can be characterized by a set of VCR features; an automobile can be characterized by a set of automobile features; or a software switch can be characterized by a set of switching features. In this scenario, there exists a software product with a large set of features from which the desired product is derived, and an impact analysis maps the desired features to the existing system. Based on the feature differences, the desired product is derived from the existing system. Currently, the derivation takes the form of a specialization of a general system, removal or addition of features, or relatively simple transformations. In this scenario, the existing or derivative system could be an intermediate artifact as well, such as an existing generic design. Features are a mechanism for comparing requirements with the existing system.

The third scenario focuses on similarities between certain large software systems and computer system software, such as operating systems and environments. Well-known models of such systems suggest a description of certain large systems as consisting of layers, each of which provides a particular class of service, and standardized protocols for interfacing adjacent layers. For example, some software systems have a user interface layer, application layer, a programatic interface layer, communications layer, control or executive layer, and resource layer. Each of these in turn may have sublayers. For example, the user interface layer may consist of presentation and dialogue layers, the communications layer may consist of the open systems interconnect (OSI) layers, and so on. A layer may also be partitioned into parts. The separation into layers and parts results in collections of services, functions or objects, and interfaces. Available parts, subsystems, layers, and protocols are formally collected, organized, and evolved for developing software for particular product areas.

The three "ideal" capabilities described here are not intended to be exhaustive nor orthogonal, but they do represent types of capabilities which have been matured for some areas. For instance, the first capability is typical of system products like compilers and interpreters, database systems, spreadsheet programs, editors, and some environments. CAMP (Common Ada Missile Parts) constructors illustrate this capability for a particular application domain: missile subsystems. The second capability is typified formally by switching products, man-machine interfaces, and generic architectures, and informally by programs used by experienced programmers in writing similar programs. Reuse of existing systems in a project family of similar systems is the basis of this capability. The third capability is typified by the structure of operating systems, compilers, communication systems, and product models. In a more rudimentary form, libraries of reusable parts, such as the CAMP components or EVB and Booch data structure parts, come under this capability. There are many other application areas for which these capabilities could be made available. Moreover, these different but related capabilities can be combined to maximize reusability.

Each capability involves the "factoring" of a goal product and the "extraction" of a factor to make it accessible for external manipulation. Reuse is realized through the factors, and different forms of reuse can be characterized by their factors. The following three diagrams illustrate the factoring and access interface for each capability.

Scenario 1

```
                          ______________
                         | standardized |
commands -------->       |  resource    |
                         |______________|
```

Scenario 2

```
                          ______________
                         | derivative   |
product    -------->     |   system     |
features                 |______________|
                                 |
                                 |
                                 v
                          ............
                          : derived  :
                          : product  :
                          :..........:
```

Scenario 3

```
Constructs:                 ...............
  layers,                   : constructed :
  subsystems, -------> :    system        :
  parts,                    :.............:
  interfaces
```

In the first scenario, the general resource is "used" to satisfy the requirements, and the operations of the general resource are "reused." In the second scenario, features of the existing system are reused, and they enable the reuse of parts of the existing system. More significantly, the process by which the existing system was developed is reused. For example, the design decisions of the derivative development are reused, subject to the constraints of the derived development. In the third scenario, the layers, subsystems, and parts are reused.

Several observations should be made. First, the factors which are reused differ, not only in their nature but also in the time they are reused. Commands are dynamic in nature; and although they may be expressed at compile time or earlier, they can be issued by a program and interpreted at runtime. Also, unlike the other two scenarios, the first scenario has no explicit goal product, which is derived or constructed. Features are characteristic of a goal product and are usually reused at requirements or specification time. System building blocks are usually reused at design and implementation or testing time. A second important observation is that these capabilities are best employed in combination. The combination provides a reuse strategy presented below. Thirdly, reuse of appropriate factors is a "practical" way of achieving some of the ideal problem solving and software development capabilities.

While the three types of reuse capabilities are related, ranking them by desirability leads to further observations. The first capability is the most desirable because it provides the highest development productivity and the most automation, and it involves the most power and breadth of application. Of the three, the third capability seems to offer the least in terms of development productivity, automation, and power and breadth of application. The first capability is available from commercial or proprietary products for a small number of application domains. The third capability is most prevalent, since most applications have or could have collections of parts to support software development for those applications. Typically, the capabilities for an application domain are likely to evolve from the third (constructed system), to the second (derivative system), to the first (commands).

4. Integration of Development Capabilities

All three types of reuse capabilities may be employed in the development. For example, a command approach applies to requirements and design, a derivative approach applies to the design, and a constructive approach to the detail design and implementation. Furthermore, the three capabilities can be combined to give an integrated approach. We view the capabilities as being evolved by a development organization, with the distinct capabilities becoming more and more compatible. Thus, as several derivative systems become available, they can be generalized to give a generic system; the derivative systems will yield candidate subsystems and parts and provide a standardized context for the parts; as the relationships between initial artifacts (derivative systems, specifications, collections of parts) become understood and tuned for an application, command and programatic access to operational capabilities of existing artifacts will evolve.

An integration of the reuse capabilities is depicted in the following diagram:

```
                                 -------------
                                 |  Generic  |
                                 |Development|
                                 |    and    |
                                 |Application|
 features ------------>          |   Model   |  -------------------
           ----------->          -------------  <--------------   |
           |                          ^                     |   |
           |                          |                     |   |
           |                          |                     |   |
           |                          |                     |   V
|------------|                        |        |-------------------|
| derivative |                        |        |layers, subsystems,|
|  system(s) |                        |        |parts, interfaces  |
|____________|                        |        |-------------------|
         ^                            |                        ^
         :                            |                        :
         :                            |                        :
         :                            |                        :
         :                            V                        :
         :        ..............................              :
         :        :       Derived System       :              :
         :        :                            :              :
         :        :    ...... control  ......  :              :
         :........:    :    :<---&---->:    :  :........:
                  :    :....:   data   :....:  :
                  :           interface        :
                  :                            :
                  :............................:
```

"Command interface" from the first "ideal" capability refers more generally to control and data information that enables a user or an external program to access the operational behavior of another program. The command interface could be in the form of an application-specific language, a message, or simply parameters. Of importance here is that the interface formally distinguish control and data aspects, as a command does, so that it clearly expresses desired operational behavior. The subboxes within the derived system box represent subsystems and layers.

The generic model is an application model which, in its most complete form, is an executable application environment for prototyping the product system development and which can be adapted to become the product itself. However, in an incomplete form, it is a high-level software architecture. The generic model includes more than an architecture. It includes constraints, implications, and decisions which can arise in analogous developments. For this reason, it is also a development model and serves as a plan for the development.

The relation of derivative systems and system constructs to the system under development (derived system) is shown indirectly (i.e., by way of the arrows to the generic model) rather than directly for several reasons:

- The generic model serves as an integrating media for the three software development capabilities and the two basic development processes, the derivative and constructive.
- It limits the number of derivative systems and constructs available during the development by imposing additional constraints. The number of derivative systems and constructs is potentially very large and will increase over time. The contributions of many of these will have already been incorporated into the generic model; thus, as the generic model is extended, the need for more derivative systems and constructs is diminished.
- The generic model provides a context that facilitates understanding for using the derivative system(s) and constructs.
- The generic model is key to formal improvement of this reuse model. The derivative systems and constructs address the project level; the generic model addresses a market area of many projects. The model can be extended to include operational capabilities for simulating and prototyping project families.
- The generic model provides a base for the formulation of a reuse strategy.

The arrow from the generic model to the collection of parts represents the influence such a model has on identifying new parts and improvements to existing parts. The arrow from the derivative system(s) to the generic application represents the abstraction of a collection of (derivative) related systems to formulate a project family, represented by the generic application model. The upward arrows from the derived system represent feedback loops for the improvement of the model indirectly via the derivative systems and the parts. The derived system has used subsystems and parts from the libraries, possibly extending them or improving them. The derived system, once completed, becomes a new derivative system for the next development.

Features are special attributes which characterize what is reusable. A user or a program manipulates features in order to reuse operations at runtime and to identify structures at derivation time or parts at construction time.

5. Generic Model Improvement

A reuse strategy should include means for improving the strategy and evolving the model. Identification of possible improvements can be accomplished through data collection activities and post mortem analyses. Evolution is facilitated by categorizing types of possible extensions. Relating the reuse strategy of the previous section to the means-end paradigm suggests several of these:

- Extending and tuning the resource, derivative, and reusable artifacts and modifications (i.e., refactoring to enhance reusability).
- Capturing constraints, design decisions, and implications which underlie the derivative system, the derived system, and the generic model (eventually improved to the point of becoming an operational model, that is, a generic system).
- Creating representations for software developments which specify artifacts, constraints, and design decisions.
- Extending the collection of available artifacts to include representations of developments and subdevelopments.
- Creating and improving mechanisms for comparing artifacts and, moreover, for comparing developments.
- Extending the generic model to allow simulation with the various artifacts and derivations that were selected and possibly modified, based on the results of the comparisons.

The mechanism for comparison and systematic extension of the generic model to allow simulation, and even prototyping, implies the necessity of domain-dependent artifacts and constraints. Domain dependencies and separation from domain independencies are likely to become increasingly difficult as improvements are made. The separation of domain dependence from domain independence is both critical and difficult. It is critical because, ultimately, that which is "ideally" reusable is that which is domain independent. It is difficult because dependence and independence are relative and subject to change with a shift in point of view and with the advance of technology.

6. Features Analysis and Initial Formulation of the Generic Development and Application Model

Domain analysis is an investigation of an application area and its application systems to determine the operations, objects, and structures which commonly occur. Domain analysis produces a definition of the domain and descriptions of objects, functions, relations, and rules which constrain them. In addition, domain analysis can include selection of a representative set of application systems to represent the domain, as well as commonality analysis to identify reusable parts for the domain. Such domain analysis is prerequisite to the development of a library of reusable parts which supports the third software development capability. A features analysis, another type of domain analysis, supports the second development capability, the derivation of a software system from an existing system. A features analysis produces feature specifications for application systems for a domain. Systems are then structured to facilitate extension, modification, and deletion of features.

Both commonality analysis and features analysis support the first development capability and the formulation of a generic development and application model. However, these analyses identify and describe; they need to be followed by a synthesis activity to produce a generic model. The generic model is not the requirements model of structured systems analysis [Gane & Sarson; McMenamin & Palmer] which tries to separate the "logical system" requirements from the "physical aspects" of the system, or the "essential activities" from an "incarnation." The generic model may, in fact, incorporate physical or incarnation aspects in order to exploit reusability. Moreover, it is a software system architectural development model. It specifies the design of a project family, i.e., a collection of systems. It is to the project family what a software architecture is to a single system. This implies a software development which has two stages: project family design and project member development. The project family design includes a high-level design of all members of the family and would be done once. Furthermore, the generic model addresses not only software products, but their development as well. It is the framework for deriving a new development and new products and for applying reusable parts. Such a generic model is the key to unifying different types of software reuse and an approach that can be refined and expanded into a reuse development methodology.

The use of a generic model for software development is a fundamental departure from traditional software development. The model encompasses solution concepts and introduces them into the development at every stage, thus violating the strong separation of phases promoted in traditional development. Indeed, software development that maximizes reuse differs fundamentally from traditional development in recognizing that the incorporation of design concepts into requirements, or implementation concepts into design, is not necessarily "bad," and it allows maximum reuse of prior developments and their products for new developments. In problem solving terms, this perspective on software reuse incorporates solution concepts into a problem description model.

7. Conclusion and Recommendation

This report has presented a perspective on software reuse that views reusability as a means for achieving—or at least approximating—three desirable development capabilities. Means-end analysis served as the unifying context for describing these capabilities. The capabilities are characterized by their initial artifacts, intermediate and goal artifacts, artifact transformations, and transformation constraints. The report reformulated these development capabilities in terms of software reusability, and proposed a generic development and application model to unify the three capabilities. The model can be formulated as a project family architecture produced from domain features analysis. It provides a framework for the development of a new project family member, as well as for the products of that development.

The approach presented in this report can be expanded to a reuse strategy for tailoring a software development methodology.

References

1. Adelson, B. and Soloway, E., The Role of Domain Experience in Software Design, IEEE Transactions of Software Engineering, Volume SE-11, Number 11, 1985, pp. 233-242.
2. Bulman, D., Model-Based Object-Oriented Design for Ada, Pragmatics, Inc. Waikoloa, Hawaii, 1988.
3. Carbonell, J.G., Learning by Analogy: Formulating and Generalizing Plans from Past Experience, Machine Learning, An Artificial Intelligence Approach, Tioga Press, 1983, pp. 137-159.
4. Gane, C. and Sarson, T., Structured Systems Analysis: Tools and Techniques, Improved Systems Technologies, New York, 1977.
5. Giddings, R.V., Accommodating Uncertainty in Software Design, Communications of the ACM, Volume 27, Number 5, May 1984, pp. 29-35.
6. Lewis, T.G., Apple Macintosh Software, Software Reviews, IEEE Software, March 1985, pp. 89-92.
7. McMenamin S.M. and Palmer, J.F., Essential Systems Analysis, Yourdon Press Computing Series, Prentice-Hall, Englewood Cliffs, N.J., 1984.
8. Rosene, F., A Software Development Environment Called STEP, ACM Conference on Software Tools, New York, NY, April 1985.
9. Schank, R.C., Language and Memory, Cognitive Science 4, 243-284, 1980.

ECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-88-TR-22	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-TR-88-023
6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INST. 6b. OFFICE SYMBOL (If applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI JOINT PROGRAM OFFICE
6c. ADDRESS (City, State and ZIP Code) CARNEGIE-MELLON UNIVERSITY PITTSBURGH, PA 15213	7b. ADDRESS (City, State and ZIP Code) ESD/XRS1 HANSCOM AIR FORCE BASE HANSCOM, MA 01731
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JOINT PROGRAM OFFICE 8b. OFFICE SYMBOL (If applicable) ESD/XRS1	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962885C0003
8c. ADDRESS (City, State and ZIP Code) CARNEGIE-MELLON UNIVERSITY PITTSBURGH, PA 15213	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO.: 63752F PROJECT NO.: N/A TASK NO.: N/A WORK UNIT NO.: N/A
11. TITLE (Include Security Classification) Perspective on Software Reuse	
12. PERSONAL AUTHOR(S) J. M. Perry	
13a. TYPE OF REPORT FINAL 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day) September 1988 15. PAGE COUNT 27 pp.
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	software reuse software development

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report presents a perspective on software reuse in the context of "ideal" software development capabilities. Software reuse is viewed as a means of achieving - or at least approximating - the ideal capabilities. A generic application and development model is proposed for unifying various types of software reuse. The model can be initially formulated as a project family architecture and produced from a domain features analysis. The approach presented in this report is intended to lead to a reuse strategy and methodology for software development.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☒	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED DISTRIBUTION
22a. NAME OF RESPONSIBLE INDIVIDUAL KARL H. SHINGLER	22b. TELEPHONE NUMBER (Include Area Code) 412 268-7630 22c. OFFICE SYMBOL SEI JPO

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

SECURITY CLASSIFICATION OF THIS PAGE