R E U S E   P R O J E C T

REUSABLE SOFTWARE ENGINEERING:

A Statement of Long-Range

Research Objectives

Technical Report 159

Peter Freeman

Department of Information and Computer Science
University of California, Irvine
Irvine, California 92717

November 10, 1980

Z
699
C3
no. 159

INTRODUCTION

This report:

1. Outlines a set of long-range research objectives that focus on the reuse of the workproducts of software engineering (e.g. specifications, designs, programs);

2. Presents some of the basic philosophy underlying the research approach that will be followed in addressing these objectives;

3. Describes a coordinated set of research efforts entitled the REUsable Software Engineering (REUSE) Project.

The major sections contain the following:

PRIMARY RESEARCH OBJECTIVE

The primary objective of this research is to develop principles, methods, and tools that permit the effective reuse of the results of software engineering.

The results of software engineering may be any of a wide group of workproducts such as an analysis, a specification, a design, a program, an integration test, or a performance evaluation. The reuse of a workproduct means that one may use it in a situation other than the original one for which it was created with less effort than would be required to create a new workproduct for use in that situation.

We call this concept reusable software engineering. Note that there are two interpretations of this phrase (reusable software engineering and reusable software engineering) which are simply different views of the same end result -- the reuse of previous efforts. The second interpretation, however, is more limited in scope in that it focuses on the reuse of software while the first connotes the reuse of a wider range of artifacts.

This is an extremely broad objective. As will be made clearer in this report, our initial efforts are directed at a much narrower objective -- the reuse of the results of analysis and design.

A fundamental hypothesis underlying this research is that present knowledge and techniques are inadequate to permit significant reuse of software engineering results. No detailed defense of this hypothesis will be presented here. However, those desiring some justification are directed toward two sources: a) Conversation with anyone involved in the creation of large software-intensive systems (ask them how much they are able to reuse previous designs); b) The book edited by Wegner (1979) in which almost all the research discussed is concerned with the initial creation of various software engineering workproducts.

It should be noted at the outset that we believe this to be a sufficiently important and difficult research objective to require a relatively long time for completion. As will be seen below, some of the specific research objectives are open-ended and thus can extend indefinitely. Even the more bounded objectives may take a number of years to reach -- even if it is possible to reach them.

The time frame that we are using in planning this research is the next ten years. That is more of an arbitrary date than one calculated from a PERT-chart. We have chosen it because we believe that within 5 years there will be substantial, but perhaps not revolutionary changes in the way that software is produced, but that within 1Ø years the pressures outlined below will be so great that revolutionary changes will be necessary.

It is the purpose of the research outlined here to help lay the groundwork for changes in the way we obtain needed software.

DEFINITION OF TERMS

The discussions in this report will be aided by the following definitions.

> software - one or more programs (usually more) that may be either systems programs (e.g. operating systems) or applications programs (e.g. accounting program).

> collection of software - any set of programs, related or otherwise; includes systems (collections in which the elements are related in particular ways).*

> lifecycle - a sequence of phases in the existence of a piece of software.

The sequence is often thought to be chronological, but may not be in practice (e.g. a system in the testing phase may be returned to the design phase if major problems arise). A typical lifecycle is shown in Figure 1; we assume that any piece of software goes through such a lifecycle, although in specific cases one or more phases may be non-existent or trivial. (E.g. Redesign of an existing operating system that is intended to preserve its functionality will probably not have an extensive analysis phase). Further discussions of lifecycles can be found in Kerola and Freeman (1980).

----

*This usage is similar to, but broader than, Belady & Lehman's usage of the same term (p. 118 in Wegner (1979)).

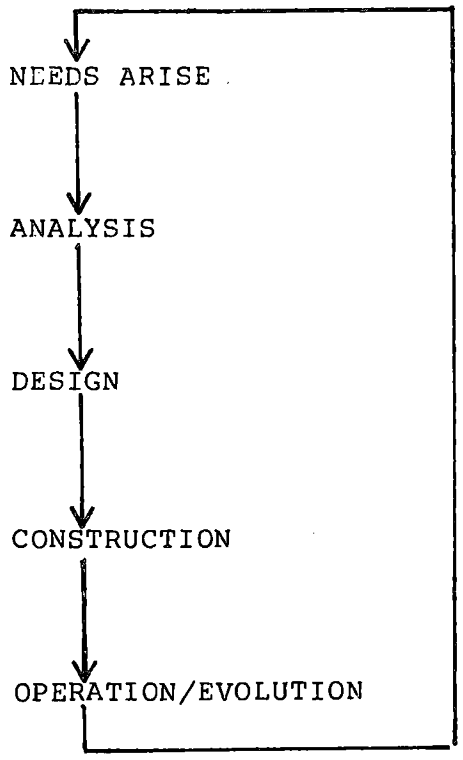

Figure 1: Typical System Lifecycle

It should also be noted that some of the phases (e.g., analysis and design) bear the name of an activity; this activity is usually dominant at that point in the life of the software, but this does not mean that the activity isn't also performed at other times (e.g., analysis and design is often done during evolution in order to adapt the software to new conditions).

<u>analysis</u> - an activity aimed at understanding a problem to be solved and at bounding the set of acceptable solutions to it.

<u>design</u> - the activity of defining and structuring a system organization in order to meet the requirements laid out in analysis. The essence of design is choice from a set of alternatives.

<u>construction</u> - in software, programming and testing (both unit and integration testing).

<u>evolution</u> - (a.k.a. maintenance) repair, adaptation, and enhancement of a software system.

<u>workproduct</u> - the result of performing an activity like analysis, design, or construction; we will always mean a tangible result that can be read and about which we can ask questions (e.g., a requirements definition

containing certain information or a design specifying the modules of a system and their interconnections).

reuse - a) use of an element of a collection of executable software (e.g. a member of library, a module fro system, a subsystem that is part of a larger sys in a new collection.

This is the conventional meaning of reuse of software and is currently practiced widely with well-defined pieces of code.

b) use of non-executable workproducts (e.g. a requirements definition, a design, a test plan) in the lifecycle of a piece of software other than the one for which it was originally produced.

This is a new kind of reuse in that it is rarely practiced at present in the software domain.

EXAMPLES OF REUSE

This section presents several examples of reusable software engineering, both current and future.

1. A subroutine used to build and initialize a particular kind of table; written and used by a programmer and a few other colleagues.

2. Mathematical subroutine package; supplied with a language processor by a vendor; used by anyone capable of using the language system.

3. A general-ledger package written in Basic and operable under the operating system of a particular vendor; built by a software house; used in many different types of business and many different machine configurations (but each of which uses the same operating system).

4. A system generator that configures an operating system for a specific hardware configuration; built by the vendor that build the operating system; usable by systems programmers for a specific type of hardware and a specific operating system.

5. An applications generator that produces report generating programs able to access a database, perform some statistical analyses, and display the results; built by a software house; usable with a specific database system, particular analysis operations, and specific display formats

6. A language system (e.g., UCSD Pascal) built in itself or some intermediate language that can be implemented on a new piece of hardware by writing a very small subset of the total system in the language of the new machine and "porting" the remainder of the system over from an existing system; built by a software group; usable (with respect to putting it on a new machine) only by experts.

7. An entire applications subsystem (e.g., telemetry data processing) that is used in toto in a new system; typically usable only by or with the aid of the people that built it originally.

All of these are examples of current artifacts that permit valuable reuse of software. The expectation (see, for example, Business Week, September 8, 198Ø) is that such reuse will expand rapidly in coming years.

These examples (which cover the range of current reusability) share several characteristics:

1. Reuse involves code; in most cases, executable code; in some, recompilable code in a higher-level language.

2. The design and analysis that went into producing the code is being reused, but only implicitly.

3. The object being reused is either completely encapsulated and thoroughly described (e.g. mathematical subroutines) or requires the intervention of an expert to permit reusage (e.g. applications subsystem).

4. No applications-level function (e.g. a particular calculation or display format) can be created (without additional programming) that is not built-in to the system.

While there are undoubtedly other examples that differ in some respects, we believe that these commonalities substantially characterize our current ability to reuse software engineering. We believe that there are other patterns of reuse that would be useful but that are impossible as long as these constraints apply. Further discussion of this point can be found in Belady and Lehman (1979) and Neighbors (198Ø).

Some examples of reusability that we would like to see include:

1. A general-purpose table builder; created by a software vendor and readily available; usable by any programmer

working in any language; simple to use and no major execution or space penalties involved in using the tables it b ilds.

This requi s the reuse of an analysis of the domain of tables and a chanism that permits the generation of efficient co and storage structures to permit implementation of specific instances. Neighbors (1980) provides a mechanism to support this type of reusability. The example would also require careful packaging and distribution of the analysis and associated mechanism; in general, that has not been done.

2. An operating system design (the functional design plus the internal logical design as represented by the modularization of the system) that could be implemented on a variety of different machines; the functional design would not change in a reuse; the modularization would not change (except perhaps for minor additions or deletions); the internal design of each module (as expressed in a PDL, for example) might remain the same or might be redesigned in response to the characteristics of the new machine.

The reuse of this design would begin at the level of the modularization. That is, once the decision had been made that the functional design was what was desired for a new machine, the implementors would begin working with the architectural design document (the modularization). Initially, they would use this design (along with other information such as the characteristics of previous implementations of this design on other machines) to determine the feasibility of implementing it on the new target machine.

Assuming the new implementation appeared feasible, it would proceed on a module-by-module basis. In cases where the internal design of a module could remain the same (a command parser, for example), the PDL could be used directly; in cases where the new machine demanded a new internal design (an I/O handler, for example), a new internal design would be done. Once all of the internal design was completed, coding could then take place.

At this point, another artifact could be reused:

3. The detailed test plan (test specifications) and construction strategy (order in which modules are built and integrated); the plans would have been prepared along with the design of the operating system and used on previous implementations.

Both of these types of reuse can be found in other disciplines. It is very common to build a house from a standard set of plans that have been modified in particular ways (for example, to fit the terrain of a particular site) and in ways that permit local variation (for example, in the choice of materials or finishes). Implementation plans are often reused; for example, computer vendors have standard procedures to follow in setting up a new computer installation.

It is clear to us that this reuse of previous software engineering efforts is needed and could save enormous amounts of effort. Yet, we know of no standard operating system design that can be purchased and implemented on a variety of machines. Yet, how many really different operating systems are there in existence today? We see very few, implying that a large amount of effort is being expended in the repetitive design of a few basic systems.

Another example is:

4. A set of software components used to construct a large system (for example, a military command and control system); the components were not built specifically for this type of system, but rather come from a supply of components used in other systems as well (for example, inventory control systems, telephonic systems, and natural-language communication programs).

The lure of software components is obvious and has been discussed in various forms for some time (see McIlroy (1968), Corwin and Wulf (1972) and Belady and Lehman (1979)). While software components are now used in productive ways, there are still serious limitations to their generality and effectiveness. Work by Neighbors (1980), however, has pointed the way to new techniques for building systems from software components; that work is described in more detail below and forms the basis for some of the current work on the Reuse Project.

MOTIVATION FOR RESEARCH ON REUSABLE SOFTWARE ENGINEERING

We assume the reader of this document is sufficiently knowledgeable in both the pragmatic world of software development and the theoretical/conceptual world of computer science to share with us a large amount of common perception. In this section we will limit our discussion to points that have motivated us to pursue this particular line of research.

Pragmatic (software engineering) Motivation

We perceive a number of pragmatic reasons motivating research aimed enhancing the reuse of prior development:

Cost. The h and ever-increasing cost of producing software is a minant theme in the computer field today. Since most of that cost is directly attributable to labor costs, anything that makes some segment of the development process less time-consuming will lower total costs. Obviously, if we can completely reuse some segment of development, then it will require much less (approaching zero) labor.

Dual nature of current reuse. There is, of course, a good deal of reuse of software engineering results today. It is our observation, however, that this reuse is either an all or nothing proposition. Entire application systems are reused with only minor changes made: A general-ledger system built for one type of business is modified slightly and works well for a different type of business; an excellent example of reuse. A message-switching system built for one application is reused for a new application with only minor changes.

At the other end of the spectrum, very small pieces of code may be reused: Mathematical subroutines, data filtering processes, functions of a 1000 different types. In the extreme, we reuse single lines of code!

While both of these types of reuse are valuable, we believe them to be insuffecent to meet the demands we see on the horizon. In particular, they do not include reuse of pieces of systems in-between the two extremes of single-function subroutines and complete systems.

The problem with reusing an entire system is that you have little flexibility (with current technology) in reusing the system for new purposes. While this is fine for some purposes, the software development situation we sion envision for 1990 will demand a much wider set of total systems than can be created in this way.

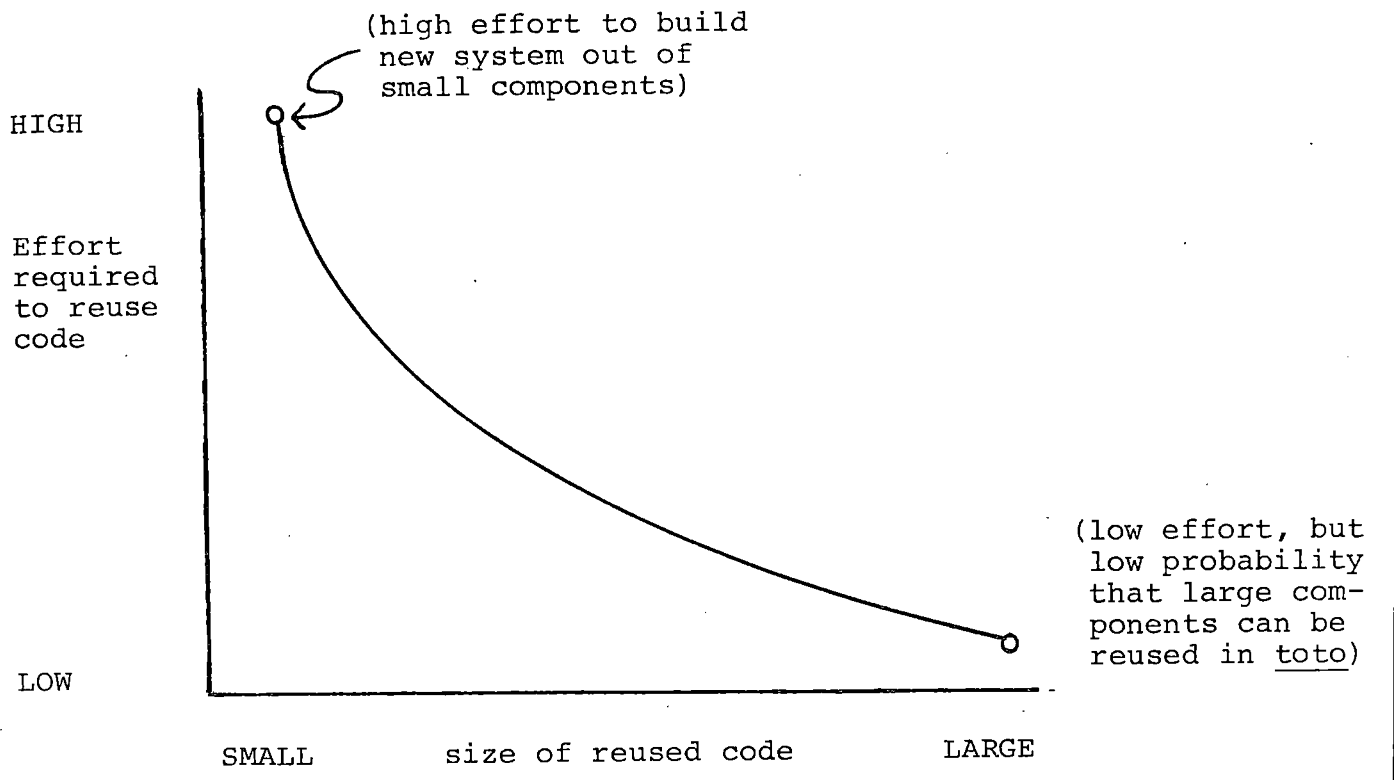

Figure 2: Reusability Tradeoffs

When we analyze the other alternative -- reuse of individual functions at the lowest level -- we also see problems. The primary one is that if one wishes to reuse small pieces of code to build a large or complex system then, in general, a large amount of effort by a sophisticated builder is needed. The result is low savings of effort. Figure 2 illustrates this tradeoff. (It should be noted that we know only the end points on these curves; their shape is unknown).

A second, and not inconsiderable, problem with the reuse of small pieces of code is that we have no pragmatic way of describing large numbers of modules so that large numbers of programmers can access them. It is as though we had a large library in which the books had no titles on the covers and there was no card catalog!

A third problem, of course, is that we have no standard ways of interfacing pieces of code (except in restricted domains limited to a particular language and usually to a particular application).

Single dimension of current reuse. A more critical problem with current reuse is that it is largely confined to the single dimension of reuse of code.

We have argued above that reuse of code alone is limited in its effectiveness due to the inflexibility of large bodies of it and the sophistication needed to use small pieces. As we learn more about how to develop systems, we increasingly understand that the harder problem is the analysis and design activity. Most people today urge that more time and effort be spent on the analysis and design for good reason -- it not only is the harder activity but if it is done properly then later stages of the lifecycle of a system, most especially evolution, will be much easier.

The motivation we take from this is that we must learn how to reuse the results of analysis and design efforts. In fact, we believe this to be the critical issue in making reusability of software engineering a reality

The advent of microcomputers. The rapid growth of capabilities of microcomputers and their rapid drop in cost is producing two main effects with respect to the need for the results of system development. First, the exploding use of small computers has only begun. One is tempted to compare it to use of electricity which started with only a few applications and rapidly expanded to its present ubiquitous role in modern society. The implication for software development should be clear. The low hardware cost and high payoff of using computers in many diverse situations will place enormous pressures on the software community to produce an incomprehensibly wide set of systems. We cannot possibly produce this multitude of applications with our current software technology.

Again, we believe that one way (but certainly not the only) to attack this is through the reuse of analysis and design. Also critical is the reuse of system parts larger than individual functions but smaller than an entire system.

Second, the obvious advantage of using microcomputers in a multitude of engineering, scientific, and data processing applications is rapidly bringing many professional engineers, scientists, and others into situations in which they must use the computer directly in order to get their job done. A current example is the design engineer on a piece of equipment. Instead of using traditional electrical systems to control the piece of equipment, increasingly he must program a microcomputer (perhaps many of them) to carry out functions previously handled by other technology.

These people are not programmers (let alone software system designers) and do not wish to become programmers. Especially because their knowledge of software is meager, they need more powerful tools. To permit them to do their job we must be able to provide concepts and tools necessary to reuse the results of someone else's problem analysis and design. In this way they will be able to reuse portions of a prior piece of software engineering work and get on with their primary jobs.

Scientific (computer science) Motivation

As will be seen when we discuss some of the specific research questions to be answered, most of them deal with how we describe and manipulate collections of programs.

Programs are fundamental. It is clear from the research that has been done and from the problems that are still unsolved that programs and the phenomena surrounding their creation and execution are the fundamental pheonomena of computer science. Most of the work in computer science so far, however, deals primarily with individual programs.

Increasingly, however, in the practical world, it is the system (of programs), not individual programs that is the element of primary concern. It creates the problems and holds out the promise of increased power. It is clear to us that computer science must learn to deal with systems just as much as it deals with individual programs.

What is relevant? What information is relevant to the description and manipulation of collections of programs? This is a basic question of computer science (when expanded to deal with systems of programs). Yet, it is one that we have no formal or organized body of knowledge about.

How are pro[illegible] systems organized? What laws govern the composition [illegible]avior, and modification of systems of programs? Agair [illegible] basic question about which we have very little informat

We will outline more fully below several questions that are fundamental to the reuse of software engineering. These questions, if answered or even partially answered, would contribute significantly to computer science. This in turn would provide a basis of knowledge applicable to many different problems.

CHARACTERIZATION OF THE REUSABILITY PROBLEM

This section provides a characterization of the reusability problem following Freeman (1976). We view reusability as basically one of description.

We assume that in a given system development situation (that is, a specific system and a specific point in the development lifecycle) an existing software engineering workproduct (such as a specification, a design, or a test plan) may be considered for reuse.* Four questions must then be answered:

1. In what ways does the existing workproduct not meet our needs?

2. In what ways can the workproduct be changed so that it does meet our needs?

3. What side-effects (unintended changes in the workproduct characteristics) will these changes induce?

4. What effects will there be on subsequent workproducts that are derived from the workproduct in question?

We call these the reusability questions.

The basic question we seek to answer then, is:

"How can a complex software system be described so that one can effectively answer reusability questions?

Operationally, there are six questions that must be answered in order to answer this overall question:

1. (definition problem) What ae the relevant characteristics of software?

*Whether or not there is such an existing workproduct -- and locating it -- is a difficult question. Here we are assuming that (somehow) a candidate has already been located.

2. (collection problem) How do we efficiently develop the information needed to answer the reusability questions?

3. (selective abstraction problem) How do we abstract only the information of interest?

4. (decision interaction problem) How do we represent the relationships between abstract aspects of the workproduct (assumptions, constraints, decisions)?

5. (system interaction problem) How do we represent the relationships (both physical and conceptual) between parts of the system that the workproduct describes?

6. (location problem) Given a specific set of requirements for a workproduct, how do we locate the information necessary to answer the reusability questions?

An additional question that carries us beyond the basic question of determining reusability, but that is essential in practice to decide if specified changes will work, is the following:

7. (changed assumption problem) How do we determine the effects of proposed changes?

There are many interpretations of these problems. We could explore each in considerable detail, considering ramifications, connections to other research problems in computer science, previous work and so on. Likewise, there are many different approaches to solving these problems that could be taken. It is one of the primary objectives of the Reuse Project to explore these and other questions in considerable detail, but we shall not do so here. The current research described later in this report does provide an indication of how we are approaching them, however.

BASIC RESEARCH QUESTIONS

It is useful to identify a small number of basic questions that underlie the large number of more specific questions that are asked in the course of a research project. In fact, the purpose of many research projects, when viewed from a perspective of time or distance, is primarily to discover what the right questions really are. Thus, we cannot state with 100% certainty that the following are _the_ basic questions; however, we have studied these areas for a number of years in different ways and always come back to the opinion that they are indeed basic questions.

Our focus is on the reuse of software engineering. To answer the questions that these goals imply, we believe that the following four, highly interrelated questions must be addressed:

> *Inform[illegible]ion question*: What information is relevant [illegible] the description and manipulation of collection [illegible]f programs? (*i.e.*, what are the relevant software e[illegible]eering workproducts?)

> *Generation/usage question*: What are the basic operations used in the creation and usage of information about program collections? (*i.e.*, how are the workproducts created?)

> *Representation question*: How should information about program collections be represented to facilitate reuse? (*i.e.*, what form should workproducts take?)

> *Structure question*: What "laws" govern the composition and manipulation of software systems?

In each of these basic questions we have used the word "programs" because that is the ultimate entity with which we are concerned. It should be clear, however, that information about program collections may include the full range of software engineering results -- designs, analyses, requirements, and so on.

These are the basic research questions that we believe our research on the reusability of software engineering will eventually help answer in part. We believe it is important to keep in mind what the underlying questions are for several reasons.

First, by being aware of the larger or more basic questions, we can more easily use results from other areas of computer science to help advance our knowledge on our specific research goal of reusable software engineering. Second, we believe that the pragmatic research question of reusability and the underlying scientific questions of software characteristics, composition, generation, and manipulation are inextricably intertwined; the pragmatic question cannot be answered without (partially) answering the basic research questions and they cannot be answered without a specific context (such as reusability) in which to explore.

RESEARCH PLAN

We do not have, nor do we feel it is appropriate to have, a completely mapped out research plan. We have been describing and bounding a research area, not a specific single-thread research project. We do have some specific, immediate research plans which are briefly described below. In this section we want to describe the general paradigm we are following to guide our long-term research activity.

Fundamentally, we believe that progress will be made through the iterative cycle of the scientific method, as shown in Figure 3. Given some concepts or theories about reusability, (derived initially from observations of and experience with attempts to reuse software engineering), we must put those ideas to the test of experimentation and actual practice. That activity, when properly evaluated, should yield information about the validity of those ideas as well as information that can be used to improve them and to generate new theories. This cycle can be seen in the immediate research plans described below.

```
 OBSERVATION
OF SE PRACTICE --> EXPERIMENTATION --> EVALUATION --> RESULTS
    ^                                          |
    |____________________<_____________________|
```

Figure 3: Basic Research Paradigm

This basic paradigm becomes more specific when we consider the nature of the phenomenon we are studying -- reusability. Figure 4 shows five aspects of the reusability phenomenon that we must study (the structure of systems, development information, methods for generating development information, methods for reusing such information, and tools to help with generating and reusing development information) and four types of research activity that (in general) must be carried out (perhaps repeatedly) on each aspect of the phenomenon.

This table can be read in several ways. Note that the column titles can be interpreted as means of providing information about the ends (system structure, definition of development information, etc.) in which we are interested. Another interpretation is that the activities are ways of helping us understand the observable artifacts of the situation. A third observation is that the artifacts become more complex as we go down in the table (i.e., methods for generating development information are more complex subjects of study than the information itself).

In the context of reusability, we believe that we must first characterize what we know about the particular aspect of the phenomenon (such as development information) and then formulate requirements on this aspect with respect to reusability. The nature of the phenomenon coupled with the requirements will then permit us to generate an experimental concept or object that can be tested through actual experimentation (and its accompanying evaluation).

A C T I V I T I E S

		CHARACTERIZE PHENOMENA	FORMULATE REUSE REQUIREMENTS	PROPOSE CONCEPT OR ARTIFACT	CARRY OUT EXPERIMENTS
ARTIFACTS	SYSTEM STRUX				
	DEVEL INFO	lc. models rep study	rep study	rep study	rep study
	GENRATING METHODS	SE theory			
	REUSE METHODS	Draco-8Ø		Draco-8Ø Freeman-76	Draco-8Ø Draco-81
	TOOLS			Draco-8Ø	Draco-8Ø Draco-81

Figure 4: Specific Research Activities

The body of this table contains specific research activities that help us address particular aspects of the problem of reuse. For example, our current research on the representation used by various development methods will primarily address the aspect of development information.

CURRENT RESEARCH

The specific research activities noted in Figure 4 are the following:

Draco-8Ø

This was an extensive piece of research resulting in the Ph.D. thesis of James M. Neighbors (198Ø). The most tangible aspect of the research is a system (Draco 1.Ø) that makes it possible to build applications systems using preexisting software components. The software components are organized into domains of knowledge about subjects (for example, data structures, natural language parsers, algebraic computations). Each domain encapsulates the operations and operands that are needed to express any desired actions in a domain; each operation and operand is implemented as a software component; the semantics of each component are expressed in terms of some other domain that Draco already knows about.

Draco then permits one to refine statements made in the language of one domain into the underlying domains used to express its semantics. (There may be more than one possible refinement of a component into another domain). Draco also permits you to optimize the resulting refinements in order to obtain the required efficiency of the resulting code.

The largest example that Draco has dealt with so far is the creation of a small relational data base system with a natural language front-end. It permitted the construction of this example system from preexisting components in a fraction of the time that it would have taken to build the system from the ground up. While this is obviously due to the fact that most of the pieces had already been built, these pieces could (and were) used in other systems as well.

Neighbors (198Ø) describes this work and also provides a number of insights into the philosophy behind the use of software components. While this work deals primarily with components, it clearly is a seminal piece of work for reusability since it permits one to reuse analysis and design as well as actual code. Its primary focus (in the reusability area) is on defining an effective method of reusing analysis and design and on providing a tool that makes this possible.

Draco-81

Work has already begun on exploring the limits of the reusability method and tool incorporated in the Draco system. This will essentialy be an experiment in which Draco is built up to a level that will permit the rapid construction of many different systems of a particular type for a variety of target machines. The system type chosen for this experiment is the class of dedicated, special-purpose relational data base systems with natural language interfaces.

Within two years we expect to be able to generate systems such as inventory control, student information, sales/order information, vehicle location, and customer information. Each system might be on the order of 1Ø,ØØØ executable instructions, run on an Apple, a PDP-1Ø, or a large mainframe, and take no more than a week to create once a complete and consistent description of the desired system is presented.

Dr. Neighbors is continuing this work under the direction of the author and with the assistance of Bruce Porter. Outside funding is being sought.

Representation Study

Work has also started on a study of analysis and design representations that is aimed at establishing a firm set of requirements for the reusability of a design or analysis and proposing a first step in the direction of an improved representation to fulfill these requirements. The core of this work will be a set of experiments to shed light on the ability of current representations to support reusability.

This work is being done by the author with the assistance of Jerry Hamilton, Ruben Prieto-Diaz, and Annette Collard. Outside funding is being sought.

Freeman-76

In Freeman (1976) we proposed a projective method of obtaining the necessary information for reusing a piece of software. We have not explored that concept further to date; however, that proposal shaped much of thought in this report and that concept still appears fruitful.

Lifecycle Models

This work, and that described in the following section, was not undertaken with reusability uppermost in mind; nonetheless, it is relevant to our work on reusability and will be continued.

The paper by Kerola and Freeman (1980) takes a first step toward analyzing the lifecycle definitions that are used to organize the software engineering activity. This is relevant to reusability since it helps us understand the points in development at which it is "natural" to define a workproduct.

SE Theory

In 1978 and 1979, in the course of working on a textbook in software engineering, we attempted to define some fundamental operations of software engineering and to describe these operations in terms of SADT models. This work has not been published nor continued in any systematic way. However, it appears to be relevant to reusability in that it helps us characterize the methods by which the information we wish to reuse is developed in the first place. This, in turn, is the first step in making sure that those methods produce information that is reusable.

CONCLUSION

We believe that the problems facing the computing community and the software profession in particular are significant when we look far enough ahead. Essentially, those problems are ones of being able to deliver on the expectations that the phenomenonal growth in computing capability are rapidly building in our society as a whole.

We also continue to be excited by the intellectual challenge of understanding the nature of this artificial science that is being created. Even if there weren't pressing pragmatic questions to be answered, the intellectual challenge could keep us busy for a long time.

We must reiterate that we in no way believe that the research area outlined here is the only answer. We do believe, however, that it is sufficiently important to warrant our primary research efforts for the forseeable future.

Finally, we want to stress the dual nature of the research outlined. The methods of reusability, when sufficiently understood and developed, should provide us tangible help in dealing with the coming flood of demands for highly varied software. For the even longer term, the focus on reusability will provide us the right orientation to study effectively the underlying nature of software and the processes surrounding its creation and usage.

ACKNOWLEDGEMENTS

A number of people have discussed reusability with me and to all of them I owe a debt for taking the time to understand, think about, and provide feedback on these ideas. Les Belady of IBM Research and Dennis Kibler of UCI were especially helpful in carefully reading an earlier draft. Jim Neighbors has long been more of a colleague than a doctoral student, and it is a pleasure to acknowledge the contribution that he has made to these ideas. Finally, Tony Wasserman, in ways direct and indirect, has again contributed to my understanding of software and the activities surrounding it.

REFERENCES

Belady, L.A. & M.M. Lehman. "The Characteristics of Large Systems," in Wegner, 1979.

Corwin, W. & W. Wulf. "SL-23Ø: A Software Laboratory," Technical Report, Carnegie-Mellon University, 1972.

Freeman, Peter. "Reusable Software", Research proposal, ICS Department University of California, Irvine, 1976.

Kerola, Pentti and Peter Freeman. "A Comparison of Lifecycles," accepted for publication in _Proc. 5th Intl. Conference in Software Engineering,_ 198Ø. (Preprint available from authors)

McIlroy, M.D. "Mass Produced, Software Components", 1968 NATO Conference on Software Engineering. Also in J.M. Buxton, P. Naur, and B. Randell (eds.) _Software Engineering Concepts and Techniques_, Petrocelli/Charter, 1976.

Neighbors, James M. _Software Construction Using Components_, Ph.D. Thesis, University of California, Irvine, 198Ø. Available as ICS TR #16Ø.

Wegner, Peter (ed.) _Research Directions in Software Technology_ MIT Press, 1979.