A Case Study in Software Reuse: The RNTDS Architecture

Barry J. Stevens
Old Dominion University

Follow this and additional works at: https://digitalcommons.odu.edu/computerscience_etds

Part of the Databases and Information Systems Commons, Software Engineering Commons, and the Systems Architecture Commons

Recommended Citation
https://digitalcommons.odu.edu/computerscience_etds/160

This Thesis is brought to you for free and open access by the Computer Science at ODU Digital Commons. It has been accepted for inclusion in Computer Science Theses & Dissertations by an authorized administrator of ODU Digital Commons. For more information, please contact digitalcommons@odu.edu.
A Case Study in Software Reuse: 
The RNTDS Architecture

by

Barry J. Stevens
B.S. (Comp. Sci.) December 1989, Old Dominion University
B.S. (Chemistry) May 1981, University of Notre Dame

A Thesis submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE
COMPUTER SCIENCE
Software Engineering Concentration

OLD DOMINION UNIVERSITY
December 1991

Approved by:

C. Michael Overstreet (Director)
ABSTRACT

A Case Study in Software Reuse:
The RNTDS Architecture

Barry J. Stevens
Old Dominion University, 1991
Director: Dr. C. Michael Overstreet

It has been asserted that the most significant gains in software productivity will come from increased levels of software reuse. Some economic models for predicting savings through software reuse exist, but none are based on actual project experience. This paper surveys those models and presents the results of a case study, the Restructured Naval Tactical Data Systems (RNTDS) architecture, in the light of those models. According to the models, software reuse via the RNTDS architecture has enabled the production of fourteen major computer programs at five to twenty-one percent of the cost to create them without software reuse. Organizational characteristics cited by the Software Engineering Institute as promoting reuse hold true for the RNTDS architecture. The high degree of software reuse in the architecture (89 to 99 percent reuse across fourteen programs) was made possible by a high degree of common functionality and from the use of design techniques which exploited it. A reusable component is not more expensive to produce a "non-reusable" one. Increased architectural and repository costs were equally offset by decreased component testing requirements. Also included in this paper are a brief survey of the software reuse state of the art and annotated and general software reuse bibliographies.
Table of Contents

1. SUMMARY

2. SURVEY OF REUSE TECHNIQUES

3. RNTDS ARCHITECTURE AND SOFTWARE DEVELOPMENT METHODOLOGY
   Combat Direction Systems
   - Motivation for a new software development architecture
   CDS Software Development in the RNTDS Architecture
   - Software Development Phases
   - Noteworthy Characteristics

4. RNTDS ARCHITECTURE IN LIGHT OF EXISTING REUSE MODELS
   Kang/Levy Model
   Gaffney/Durek Model
   Gaffney Model

5. CONCLUSIONS

ANNOTATED BIBLIOGRAPHY

ADDITIONAL REUSE REFERENCES
LIST OF TABLES

TABLE 1  Cost matrix with groups A and B 22
TABLE 2  Cost matrix with groups A and B with user of software paying X 22

TABLE 3  Relative Project Cost, $C=(b-1+\frac{E}{n})R+1$, From Gaffney/Durek model 27

TABLE 4  Relative Project Cost, $\frac{C_{US}}{C_{UN}}$, From Gaffney model 29
<table>
<thead>
<tr>
<th>FIGURE</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIGURE 1</td>
<td>Combat Direction System Functional Commonality</td>
<td>14</td>
</tr>
<tr>
<td>FIGURE 2</td>
<td>RNTDS Architecture Software Development Phases</td>
<td>15</td>
</tr>
<tr>
<td>FIGURE 3</td>
<td>Derivation of $C_{UN}$ and $C_{UR}$ for RNTDS Architecture</td>
<td>28</td>
</tr>
</tbody>
</table>
1. SUMMARY

Survey of Reuse Techniques

Current software reuse techniques are categorized as follows:

- Classification techniques
- Design techniques
- Code development techniques
- Software engineering environments
- Techniques for programming-in-the-large
- Techniques for program transformation

Examples of each technique are cited from the literature.

RNTDS Architecture and Software Development Methodology

Combat Direction Systems are a diverse family of mission critical, real-time command and control systems in use today aboard Navy ships. Current CDS computer programs contain over 250,000 source lines of code which support operator display and interaction and several intercomputer and peripheral interfaces with stringent timing requirements.

A new software architecture, named Restructured Naval Tactical Data System (RNTDS) was developed to capitalize on a high degree of functional commonality across various CDS programs by introducing a formal reuse paradigm. The goals of RNTDS include: the automated construction of multiple CDS programs from a single reuse repository, the reduction of life cycle maintenance costs, the ability to change multiple programs through a single implementation, and the creation of a common operator interface across all CDS programs. CDS program development under RNTDS has been guided since 1982 by MIL-STD-1679A. The major activities involved in RNTDS program development are summarized. The RNTDS architecture implements many of the reuse techniques discussed in Chapter 2.

RNTDS Architecture in the Light of Existing Reuse Models

Software reuse, as implemented by the RNTDS architecture, enabled Combat Direction System computer programs to be produced as low as 5 (by Gaffney model) to 12 (by Gaffney/Durek model)
percent of the cost to produce them without reuse. This translates to a 8 to 20-fold productivity increase.

The organizational characteristics cited by the Software Engineering Institute (Kang and Levy) as promoting reuse hold true for the RNTDS architecture.

The high degree of reuse in the RNTDS architecture (89 to 99 percent across fourteen distinct programs) was made possible by the high degree of common processing functions required by the fourteen programs and from the use of design techniques, such as the uses of abstraction and small Ada-like components. Consequently, projects which foresee similar functional commonality and organizational characteristics may see similar benefits from reuse.

The benefits of reuse increase as more phases of software development are impacted. This effect is captured by the Gaffney/Durek parameter $b$.

A reusable component is not more expensive to produce than a "non-reusable" one. The increased unit cost due to the creation of a reuse repository and software support environment is offset in the RNTDS architecture by a decreased testing requirement. This effect is captured by the Gaffney/Durek parameter $E$.

The Gaffney/Durek and Gaffney reuse economic models properly demonstrate that software reuse is a significant enabling technology toward improving software development productivity.

More empirical information, specifically project reuse experiences reported in terms of the models, is needed to make the models more useful for projects which are still in the planning stages.

The models provide, as reported in Tables 3 and 4, comparisons of project performance against no reuse. A more meaningful contrast would be an economic comparison among various levels of reuse. Such a hypothetical comparison, within the framework of the models, is a potential area for further study.
2. Survey of Reuse Techniques

This section surveys the following types of reuse techniques:

1. Classification techniques: How to organize a software repository
   a. Faceted classification
   b. Domain analysis

2. Design techniques
   a. Information hiding
   b. Object oriented design

3. Code development techniques: Language selection

4. Software Engineering Environments - Tools needed to support reuse
   a. Tools for design and production
   b. Tools for program testing
   c. Tools for life cycle maintenance/program adaptation
   d. Tools for entire development cycle

5. Techniques for Programming-in-the-Large
   a. Module Interconnection Languages
   b. Library Interconnection Languages

6. Techniques for Program Transformation
   a. Instantiation of design abstractions
   b. Software reengineering

Classification Techniques

Classification techniques attempt to describe how software components should be tagged with identifying information so they can be located and retrieved easily. In some ways this is similar to the problem of classifying books in a library; a wide variety of subjects are covered, yet books on specific subjects can be found.

Ruben Prieto-Diaz and Peter Freeman draw from library science in their proposal of a "faceted" classification scheme which is similar to the Dewey decimal system [Prieto-Diaz87]. Although several papers described a query system of identifying software components by information stored in n-tuplets, the Prieto-Diaz/Freeman proposal was most complete, addressing such issues as vocabulary control, metrics for evaluating the reusability of components and methods for assessing the effectiveness of a classification scheme.

Domain Analysis is a technique of breaking down an application domain into standard com-
ponents, which becomes a standard for organizing a reuse library for that domain. A detailed example of a domain analysis is provided by Shlaer and Miller [Shlaer89] in which the processes involved in a fruit juice processing plant are decomposed in an object oriented manner into what could be a standard for describing all such plants and for cataloging their processes. The technique is appropriate for object or functionally oriented designs. Domain analysis reinforces the idea that repositories need not be universal in scope, but may serve single application domains. One step-by-step method for conducting a domain analysis is presented by the Software Engineering Institute (SEI) in [Kang90]. An SEI domain analysis bibliography is published as [Hess90].

Design Techniques

A general agreement seems to be that if a reuse paradigm is to be used, the design phase of software development needs to be impacted. Veikko Seppanen [Seppanen88] and T. Moineau [Moineau88] highlight the general need for including reuse as a design goal and for storing design information with the component in a reuse repository. Specific design techniques which emerge from the literature are information hiding and object oriented design. The additional design topics of abstract data typing and canonical design are addressed under Transformation Techniques.

Software cost reduction, not reuse, was the original motivation for the SCR project of D. L. Parnas. This effort popularized the concept of information hiding, a design technique through which software costs can be significantly reduced by keeping software changes as localized as possible. Later, Parnas describes how information hiding enhances reusability [Parnas83]. More recent corroboration comes from [Hagar89] and [Gingerich89].

When object oriented design was described by Grady Booch [Booch83], its impact on software reuse was not directly addressed. Later, the two concepts became tightly linked ([Booch87] and [Somerville89]). Much current literature exists which provides examples of reuse projects based on object oriented design; the two concepts have helped to popularize each other. It seems that the principal benefit of object oriented design toward furthering reuse is to encourage the description of processes in an abstract manner, so that neither the actual data structure used to implement an object nor the algorithms used to manipulate the objects are visible to the surrounding processes. Object oriented design is
a logical extension to the small component or task level of information hiding techniques described for large modules.

**Code Development Techniques**

The selection of an implementation language will impact the reusability of the product. Some languages are cited as enhancing reusability:

- Ada [Booch83], [Booch87], [Wegner83].
- Objective-C, a C extension [Love88], C++
- Smalltalk-80 [Deutsch83], and
- Lisp [Wegner83], all for support of object oriented programming.
- Unix [Kernighan84], for support of function libraries, the pipe mechanism, hardware independence and program combination through input/output redirection. Unix may be considered both a support environment and, when using shell scripts, an implementation language.

**Software Engineering Environments**

General agreement exists in the surveyed papers that new software tools are needed to support reuse. Some tools support specific phases of software development: design/production or maintenance, while others address the entire software development life cycle.

A few software environments are described in the literature which assist in implementing reuse in the design and production phases of development:

- The REUSE [Arnold87] and MOMOCLI [Tarumi88] projects are representative examples of several repository and component retrieval systems described in current literature. In general, these tools support the location of components through keyword searches or other query techniques, their retrieval, addition, adaptation, deletion and assistance in interfacing the components in new combinations.
- Wlisp [Fischer87], a knowledge-based tool which supports design; and, again,
• Unix [Kernighan84], which supports software design and production through the features described above.

Based on the premise that software maintenance or adaptation is a particularly intense form of reuse, two projects have taken aim at the problem of support tools for life cycle maintenance:

• The Demeter project [Lieberherr88], which "grows" reusable objects from the "seeds" of a data dictionary; and the

• SMSE (Software Maintenance Support Environment) project [Wild88], which is beginning to identify the support requirements for program maintenance in a reuse context and to identify technologies to meet them.

While not mentioned in the cited literature, support of reuse in program testing exists. Reuse of program test cases is accomplished in a variety of CASE tools which help generate test cases and store them for future use. Reuse of test cases is beneficial in two ways: time required for test case generation is saved, and repeated use of the same test cases is a means of regression testing.

STARS (Software Technology for Adaptable Reliable Systems) [STARS90] is a Dept of Defense sponsored project with a goal of "supporting modern reuse-oriented software processes with open architecture software engineering environments" for the entire software life cycle. Through STARS, the efforts of three prime contractors are currently being coordinated to provide integration of existing "off the shelf" commercial tools, tools to support formal proof methods, and a demonstration domain-specific repository. Multi-vendor projects such as STARS and the Software Productivity Consortium efforts also increase the likelihood that multi-user repositories will become more widespread.

**Techniques for Programming-in-the-Large**

*Programming-in-the-Large* refers to the practice of constructing large, complex computer programs from small reused components and some new ones. Most efforts along this line address the technique of specifying a new system by specifying its overall structure and the requirements for inter-module interconnections. Four *Module Interconnection Languages* (MILs), which are formal grammar constructs for assisting programming-in-the-large are surveyed in [Prieto-Diaz86]. Related work is
described by [Rice83] and [Wolf85].

J. Goguen [Goguen86] maintains that while MILs provide a "formal grammar construct to decide the various module interconnection specifications required to assemble a complete software system," Library Interconnection Languages (LILs) are required to describe what each software component is required to do in terms of its interior design. In his paper, an Ada-like specification language is proposed.

Techniques for Program Transformation

Program transformation techniques are generally based on the premise that the reuse process should begin with (1) identifying actually or potentially reused components, then (2) decomposing or abstracting those components into a generic form for cataloging and later use [Boldyreff89].

A frequently cited effort of this type is the Draco project [Neighbors87], which attempts to assist reuse by "capturing the results of analysis and design in a way which facilitates reuse." The process involves a series of transformations from specification to code. Other similar efforts describing the instantiation of generic or abstracted components is found in the PARIS project [Kate87] and research described in [Embley87] and [Harandi88]. The Software Productivity Consortium has sponsored related research in canonical design, a technique based on accessing a library containing similar components using generic schema [Pyster88].

Software reengineering claims that transformations need not begin with stored generic components. In an effort described in [Sneed87], a somewhat unstructured and heavily modified COBOL-74 application was reverse engineered, modularized, adapted and incarnated as an equivalent COBOL-85 program. The motivation for this effort was to salvage reusable portions of an old program not initially designed to be reused. Considerable research is underway in this area. [NISQP91] and [NSWC91] are proceedings from recent conferences on this topic.
3. RNTDS ARCHITECTURE AND SOFTWARE DEVELOPMENT METHODOLOGY

This section provides an overview of the software development activities and methods used in the production of Combat Direction System programs.

3.1. Combat Direction Systems

The term Combat Direction System (CDS) applies to a diverse family of mission critical, real-time command and control programs in use today aboard Navy ships. Formerly known as Naval Tactical Data Systems, these programs were introduced over twenty-five years ago in response to the increasing numbers and speeds of potentially hostile aircraft and cruise missiles. Shipboard decision makers and weapon system operators became required to identify, track and respond to many high speed objects in the air around them, a task which overwhelmed the slow, manual methods in use at the time. The need for automation to support the real-time decision making and reaction requirements of battle was identified.

CDS programs for today’s non-Aegis cruisers and guided missile destroyers are hosted in the dual-CPU AN/UYK-43B tactical computer. The software contains over 250,000 source lines of CMS-2 code and performs various real-time processing functions. It interfaces with several sensor, weapon, and communication systems throughout a ship and supports consoles through which operators see tactical information and program-generated recommendations. Operators enter orders into consoles which are carried out by the program or passed to the appropriate system for execution. Digital data communications via radio are used to link CDS programs to permit the transfer of information and orders among ships which are working together.

The Motivation for a New Software Architecture

The Navy requires a number of different ship classes with various combinations of sensors and weapons to meet specific requirements. The impact upon CDS programs is that while some software requirements may be common across all or many ships, others may be unique to one ship class. A summary of functional commonality is presented as Figure 1, which shows that, despite hardware differences, 60 to 80 percent of the processing requirements were expected to be common. The
RNTDS architecture exploits that commonality.

<table>
<thead>
<tr>
<th>EXAMPLES OF FUNCTIONAL AREAS IN WHICH ALL OR MAJORITY OF TASKS CAN BE COMMON TO ALL SHIP CLASSES</th>
<th>FUNCTIONAL AREAS UNIQUE TO SHIP CLASS</th>
<th>FUNCTIONAL AREAS UNIQUE TO COMPUTER OR HARDWARE</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA BASE</td>
<td>TRAJECTRY</td>
<td>EXECUTIVE</td>
</tr>
<tr>
<td>DISPLAY1</td>
<td>THREAT EVALUATION</td>
<td>SYSTEM CONTROL</td>
</tr>
<tr>
<td>DATA EXTRACT</td>
<td>WEAPON SELECT</td>
<td>PERIPHERAL CONTROL</td>
</tr>
<tr>
<td>DATA REDUCTION</td>
<td>WEAPON ASSIGN</td>
<td>SIGNAL DATA CONVERTER</td>
</tr>
<tr>
<td>AIR CONTROL</td>
<td>TEST CONTROL</td>
<td>KEYSET CENTRAL</td>
</tr>
<tr>
<td>LINK 6A</td>
<td>AUTOMATION</td>
<td>MULTIPLIER</td>
</tr>
<tr>
<td>ELECTRONIC WARFARE</td>
<td>RADAR VIDEO</td>
<td>UTILITY PACKAGE</td>
</tr>
<tr>
<td>IDENTIFICATION</td>
<td>AUTOMATIC DETECT</td>
<td>DATA TRANSMISSION</td>
</tr>
<tr>
<td>NAVIGATION</td>
<td>VAVE TRACK</td>
<td>NAVIGATION</td>
</tr>
<tr>
<td>MILIPERATOR</td>
<td>SATELLITES</td>
<td>ELECTRONIC WARFARE</td>
</tr>
<tr>
<td>OMEGA</td>
<td>MAP DISPLAY</td>
<td>ELECTRONIC WARFARE</td>
</tr>
<tr>
<td>SURFACE OPS</td>
<td>PEARL M. S.</td>
<td>ELECTRONIC WARFARE</td>
</tr>
</tbody>
</table>

Figure 1 - Combat Direction System Functional Commonality

In older architectures, each of the functions were implemented by fairly large (1 - 5 KSLOC) modules. Common functions were implemented across the various ship classes by several similar modules, each slightly changed to accommodate the unique hardware configuration or other specific processing requirement for that program. Improvements required for all ships needed to be individually designed and implemented for each class. The problem was compounded by the fact that not all ships in a class are identical, causing processing differences to exist within the same ship class.

In the late 1970's the Restructured Naval Tactical Data System (RNTDS) architecture was developed at FCDSSA Dam Neck with the following goals:

- Construction of multiple CDS programs with varying requirements from a single repository of small, reusable components. After the first program has been implemented, remaining programs should be built for significantly less cost.

- Reduction of life cycle maintenance costs through the abstraction of common processing requirements. Reduction of the impact of software change resulting from corrections, improvements or hardware changes. This goal was influenced by the work of D. L. Parnas [Parnas72].

- An ability to deliver CDS program improvements across all ship classes via a single implementation.

- A commonality of operator interfaces across all ship classes to minimize training requirements.

In 1982, the CG/DDG CDS Upgrade project began which improved the CDS programs for seven
ship classes and rehosted them into their current computer. The RNTDS architecture was used for the multi-program project.

3.2. CDS Software Development in the RNTDS Architecture

This section is an overview of the RNTDS software development methodology.

3.2.1.

There are over thirty-five separately scheduled activities involved in RNTDS program development, each with a defined process, end product deliverables and applicable metrics. At a higher level, software development in this architecture can be described in terms of general phases which are organized in a waterfall-type model. Figure 2 depicts the relationship among development phases with critical reviews and milestones.

![Figure 2 - RNTDS Architecture Software Development Phases](image-url)

The remainder of this section will discuss the activities which occur in each phase. The references to database relations in the following activities refer to the RNTDS integrated database ADVISE (ACDS Database Virtual Systems Engineering).

CDS "A" Specification Development

The CDS "A" Specification is a high level description of Combat Direction System functions. It is the means through which the Naval Sea Systems Command project sponsor conveys program requirements to FCDSSA Dam Neck for implementation. The specification allocates requirements to hardware and software. It is baselined and put under configuration control at the Systems Requirements Review
Program Performance Specification (PPS) Development

The PPS specifies the performance requirements of each Combat Direction System program. Expressed in natural language, standardized syntax and strict formatting are used to ensure clarity and to permit basic consistency and completeness checks. Program requirements are organized by functional area. Within each functional area, each indivisible requirement is expressed as a numbered paragraph. Stimulus-response processing threads are followed by following links throughout the paragraphs.

The requirements paragraphs are supplemented by information contained in the following appendices:

Appendix A: Applicable documents. Higher level requirements.
Appendix B: Glossary of terms
Appendix C: Algorithms and constants
Appendix D: Input/output formats and ranges of valid values
Appendix E: Data extraction or breakpoint requirements
Appendix F: Specifies which paragraphs, figures and tables comprise the performance requirement for each program. It is maintained as an ADVISE database relation.

Reuse begins here. Most (89 to 99 percent) of the requirements for a new program have already been written and implemented for a previous program. Some of the new requirements are written as variants of existing ones while other paragraphs are written from scratch. Since Appendix F links each PPS paragraph to the programs in which it is used, a single textual change to a requirement paragraph effectively changes the requirement for all applicable programs.

Program Performance Specification development begins with the identification of requirements to be reused, modified and created from scratch in order to support the new program. At the conclusion of PPS development it is formally reviewed at a System Design Review (SDR) and placed under configuration control.

Preliminary Design: Design Analysis Report (DAR) Development

The Design Analysis Report provides high level system flow diagrams for new capabilities and identifies the software components (CMS-2 tasks) which need to be reused, modified or built from
scratch to implement the program requirements specified by the PPS. Consequently, the DAR provides a well-refined estimate of software size for the new program, from which revised cost and schedule estimates are made. Various database relations at the preliminary and detailed design levels maintain traceability from PPS requirement paragraph to its source code implementation. At the conclusion of preliminary design a Preliminary Design Review (PDR) is held and the DAR is baselined and placed under configuration control.

Pre-Code Analysis

The activities in this phase take the preliminary design information contained in the DAR and produce a skeleton system tape. Such a tape consists of all the reused software components and stubs for the new and modified ones. An engineering workbook is created for each new or modified component which, at this point, contains the entries required to be added to the various ADVISE relations to support this component in the architecture. Formatted task preambles, which appear as comments at the beginning of each task, are constructed. Compilable stubs are created for each new task. A skeleton system tape, which is a model of the eventual program, is produced from the reused components and stubs.

The motivation for producing a skeleton system tape before new code production is to permit early resolution of problems relating to the mapping of the entire program into 8K-word segments of memory. Each software component requires certain data elements and certain other components to reside in the same 8K memory segment to optimize runtime performance. Those dependencies are documented in the preamble for each task. During this phase, the dependencies are reconciled and a blueprint for fitting the components together, called a configuration control file, is produced and a skeleton system is generated. The automated process through which library elements are brought together to form CDS programs is detailed in [FCDSSADN91].

Detailed Design, Code, Compilation, Unit Test and Integration Test

This phase is concerned with the software components which need to be built from scratch or modified. The components which were reused without modification are already on the skeleton system
Detailed design is the design of the logic within the software component. It is expressed as a flow chart which becomes part of the engineering notebook for that component. Coding, in CMS-2Y, and compilation proceed in accordance with architecture coding standards. Unit test is conducted by originator and supervisor code walkthroughs and a formal peer review. An independent Navy review is also conducted. Machine testing is available for the verification of internal component logic. Critical Design Review (CDR) is conducted at the conclusion of unit test. Finally, integration test exercises all new stimulus-response paths in a low system stress environment. This testing verifies the correctness of inter-component communication as well as internal component logic.

Function Test

After the baselining of the PPS at the System Design Review and during program design and production, the Navy writes Functional Test Procedures (FTPs) which are based on the PPS requirements. During this phase, the new program is tested against the FTPs. FTPs are the means of ensuring complete requirements coverage of the testing in this phase. Failure of the program to perform as expected by the FTPs are documented as program (or FTP) trouble reports. After correction the FTP is reexecuted. The Operators Manual for the new program is verified during Function Test. Program endurance tests are also conducted during this phase to ensure a minimum program endurance of eight hours. At the conclusion of Function Test, a Test Readiness Review (TRR) is held.

Performance Acceptance Test (PAT)

This phase is the Navy's opportunity to certify that the new program meets the functional requirements specified in the PPS and the endurance requirement (25 hours) for fleet delivery. PAT consists of a mix of FTP execution, and operationally realistic scenario testing.

Combat Systems Integration Test (CSIT)

The Combat Direction System is only one part of the total shipboard combat system. Other combat system components perform such functions as radar processing, weapons direction, intelligence
gathering, navigation or mission planning. Combat Systems Integration Test is the Navy’s opportunity to test the performance of all combat system components before shipboard delivery.

**System Integration Test (SIT) and Fleet Introduction Support**

This is the phase where the new Combat Direction System and other new combat system components are installed aboard a Navy ship. Testing during this phase focuses on program interfaces to ensure that all hardware interconnections have been properly made. Fleet Introduction Support refers to the two to six month period following SIT during which program trouble reports are corrected and the program is transitioned to life cycle maintenance.

### 3.2.2. Noteworthy Characteristics of the RNTDS Architecture

The RNTDS architecture implements many of the reuse techniques described in Chapter 2.

The classification technique used to organize the repository is based upon the program requirements in the Program Performance Specification. Strict traceability between requirement paragraphs through program design to their implementation as executable components is maintained. Essentially, applicable reusable components are selected automatically when the reused requirements for that program are selected.

The RNTDS architecture has drawn heavily on the design principle of abstraction. Application logic was made as device independent as possible. An original motivation for the architecture was to localize program changes to minimize life cycle maintenance cost.

A comprehensive software engineering environment was built to facilitate reuse throughout each phase of software development. The support tools include the means of accessing and reusing software requirements, design, compiled code and test procedures through the unified relational data base management system known as ADVISE. Data within ADVISE is used to automatically extract needed logic and data elements and to piece them into an executable computer program in accordance with the respective data and size dependencies of the components. This is also an implementation of programming-in-the-large in that reused components are fitted together at the component level by looking only at external dependencies and not interior logic.
The RNTDS software development processes are highly defined and repeatable, a process attribute which enhances reusability of the product. Processes are defined through Navy guidelines such as [FCDSADN91] and through a comprehensive work breakdown structure.

Managerial planning and monitoring processes are also highly defined. A standard set of metrics are used, as applicable, throughout the development cycle. Formal automated models are used for software sizing, schedule and cost estimates. The cost model, developed by the author, includes a means to collect and use activity costs to update model parameters and improve accuracy. The means to quantitatively monitor continuous improvement of RNTDS development processes has been established. Those means include assessments against the Software Engineering Institute five-level software development maturity model.
4. RNTDS Architecture in the Light of Existing Reuse Models

Software reuse economic models attempt to answer one or more of the following questions [Pfleeger91]:

1. How do we decide when the benefits of reuse outweigh the costs?
2. How do we estimate the cost of making a component reusable?
3. How do we estimate the benefits of reusing a component?

This chapter examines the RNTDS architecture in terms of three published software reuse economic models:

- Kang/Levy, from Software Engineering Institute
- Gaffney/Durek, and
- Gaffney, both from Software Productivity Consortium.

4.1. Kang/Levy Models

When beginning a software development project, planners select the desired software development methodology from a number of options. Such options include:

- Use of object-oriented development
- Use of expert systems
- Use of metaprogramming, or "programming at a higher level of abstraction and developing a translator to produce the nominal source code," and includes applications generators.
- Software reuse

Kang and Levy [Kang89] maintain that the selection of a successful methodology must be founded on sound technical and economic bases. We know that the above three options are each technically viable in some domains; the important contribution of this work is to provide quantitative models for these techniques which express the relationship of various pertinent factors. Each model attempts to disclose the conditions under which each particular technique may be cost beneficial. The reuse model also provides a list of organizational and application attributes which enhance the ability to successfully implement reuse.
• Use of a matrix organization where the same applications experts support multiple projects.

Finally, the authors support, under certain conditions, the creation of a central support organization. Such an organization performs the following software development functions: collect reusable components from developers, make the components available to other users, correct errors, enhance features and identify the need for new components based on new requirements.

4.1.3. RNTDS Architecture in the Light of Kang/Levy Reuse Model

4.1.3.1. Organizational Issues

The four characteristics of organizations achieving a high level of reuse hold true for the RNTDS architecture:

• All fourteen applications were within the same well defined domain

• The majority of the development (with the exception of some software support tools and the compiler) took place at two sites about fifteen miles apart

• The RNTDS architecture imposes programming conventions which support reuse; and

• A modified matrix organization is in use at the government software development facility.

A central support organization was also implemented within the Navy organization. After development and acceptance, reusable components are stored in the RNTDS Common Reusable Library (CRL). The control group for this repository provides the means for accessing the library and is responsible for Life Cycle Maintenance (error correction and minor enhancement after program deployment). The only departure from the Kang/Levy description of a central support organization is that the need for new components is documented by the development project team rather than by the CRL control group.

4.1.3.2. Comments on the Kang/Levy models

1. Kang and Levy describe their models as "back of the envelope" and are intended to provide an intuitive feel for how more quantitative models may be developed. High level models such as these provide a useful framework for communication. It is hoped that reports of actual project experience
may form the basis of real quantitative models from which savings through reuse can be predicted.

2. The array of models presented by Kang and Levy could be extended to include other software development methodologies such as prototyping and use of formal methods. Such an extension would move the software engineering community closer to a compendium of best practices. Such a compendium should include economic bases for each technique, as proposed by Kang and Levy.

4.2. Gaffney/Durek Reuse Models

Three reuse and productivity models have been developed by Gaffney and Durek [Gaffney88] for the Software Productivity Consortium. The three models present successively more detailed views:

(1) Simple model, in which the cost or savings of reuse is shown, where

\[ C = \text{relative cost to develop a product (Cost with no reuse = 1)}, \]
\[ R = \text{proportion of reused code in the product (R \leq 1)}, \]
\[ b = \text{cost to incorporate reused code, relative to cost of building components from scratch (b < 1, hopefully)}. \]

Thus, \( C = (1-R) + (R \times b) \), or the relative cost to develop a product equals the relative cost of new code plus the relative cost of reused code.

More conveniently:

\[ C = (b-1)R + 1. \]

The productivity increase or decrease in each of the models is given by \( \frac{1}{C} \).

(2) Cost-of-development model, augments the simple model by representing the additional cost of developing components suitable for reuse, where

\[ E = \text{cost of producing reusable code relative to the cost of producing non-reusable code. Gaffney and Durek expect E > 1. Values of 1.25 and 2.0 have been suggested by [Lenz87] and [Gaffney88], respectively.} \]

\[ n = \text{number of uses over which the code cost is amortized (1 use and n - 1 reuses)}. \]
Thus, \( C = (1 - R) + R \left( b + \frac{E}{n} \right) \), or more conveniently,

\[
C = (b - 1 + \frac{E}{n})R + 1.
\]

The number of reuses needed to break even (meaning that \( C \leq 1 \)) is a function of \( E \) and \( b \), specifically

\[
n_0 = \frac{E}{1 - b}
\]

(3) General model, is an extension of the Cost-of-development model which treats the generalized case where code is reused among multiple repositories.

4.2.1. RNTDS Architecture in the Light of Gaffney/Durek Model

In this section the results of fourteen RNTDS program development efforts are examined in terms of the Gaffney/Durek cost-of-development model:

\[
C = (b - 1 + \frac{E}{n})R + 1
\]

Estimating \( b \)

The parameter \( b \) is

\[
\text{Cost to reuse a component} \quad \frac{\text{Cost to build a component from scratch}}{\text{Cost to build a component from scratch}}
\]

Gaffney and Durek suggest that values for \( b \) can vary widely depending upon how broadly reuse is adopted throughout the project. If, as they estimate, requirements, design, implementation and testing constitute 27, 50, 15 and 8 percent of the total development cost, then a project which adopts reuse only in the implementation phase may see a \( b \) value near 1 - 0.15 or 0.85. Conversely, if the requirements, design and implementation phases are using reuse techniques and the testing phase is not, then \( b \) may approximate 0.08. In the RNTDS architecture, reuse is implemented in all software development phases.

Estimating \( b \) for the RNTDS architecture can be accomplished as follows:

\[
\text{Cost to incorporate a reused component} \quad \frac{\text{Cost to build a component from scratch}}{\text{Cost to build a component from scratch}}
\]
Based on project data, $b$ is calculated to be

$$\frac{8.1768 \text{ hrs per reused component}}{351.4548 \text{ hrs per new component}} = 0.0233$$

Estimating $E$

$$E = \frac{\text{Cost per unit to develop a reusable component}}{\text{Cost per unit to develop a non-reusable component}}$$

This equates to the sums of the "per component" costs for the following activities:

\[
\begin{align*}
\text{Requirements} + \text{Design} + \text{Code/debug} + \text{Testing} + \text{Arch/repository cost for reusable component} \\
\text{Cost per unit to develop a non-reusable component}
\end{align*}
\]

Based on a recent project study, this is shown to approximate 1.0. Although reuse brings with it an increased architecture/repository cost, there are equal savings in testing. The testing savings arose from the fact that detailed stimulus-response testing for a large module needed to be repeated when modifications were made to it. In the RNTDS architecture, where there are small, abstracted components, this testing was only required for the new and modified components which were produced.

Estimating $n$

Measurements of the amount of component reuse were available for eight of the fourteen projects: D, E, F, G, I, J, L and M. The data was treated as a representative sample of reuse distribution over the fourteen projects. Of the 4,912 repository components analyzed:

- 675 were used in 1 project
- 240 were used in 2 projects
- 205 were used in 3 projects
- 154 were used in 4 projects
- 185 were used in 5 projects
- 44 were used in 6 projects
- 30 were used in 7 projects
- 3,379 were used in 8 projects

A weighted average calculation for the eight projects shows that each component, on average, was used in 6.28 projects. Normalizing for fourteen projects, $n$ is estimated as 11.

Computing relative cost for each project

Assuming that $b = 0.0233$, $E = 1$ and $n = 11$, the following table shows the cost of each RNTDS project relative to the cost of development without reuse. For projects which are composed totally of
reused code \((R=1.00)\) then \(C = \frac{E}{n} + b = 0.114\).

<table>
<thead>
<tr>
<th>Project</th>
<th>(R) (% reused code)</th>
<th>(C) (Cost relative to no reuse ([R = 0]))</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0%</td>
<td>1.00</td>
</tr>
<tr>
<td>B</td>
<td>95%</td>
<td>0.16</td>
</tr>
<tr>
<td>C</td>
<td>89%</td>
<td>0.21</td>
</tr>
<tr>
<td>D</td>
<td>97%</td>
<td>0.14</td>
</tr>
<tr>
<td>E</td>
<td>99%</td>
<td>0.12</td>
</tr>
<tr>
<td>F</td>
<td>99%</td>
<td>0.12</td>
</tr>
<tr>
<td>G</td>
<td>99%</td>
<td>0.12</td>
</tr>
<tr>
<td>H</td>
<td>99%</td>
<td>0.12</td>
</tr>
<tr>
<td>I</td>
<td>89%</td>
<td>0.21</td>
</tr>
<tr>
<td>J</td>
<td>93%</td>
<td>0.17</td>
</tr>
<tr>
<td>K</td>
<td>99%</td>
<td>0.12</td>
</tr>
<tr>
<td>L</td>
<td>99%</td>
<td>0.12</td>
</tr>
<tr>
<td>M</td>
<td>99%</td>
<td>0.12</td>
</tr>
<tr>
<td>N</td>
<td>99%</td>
<td>0.12</td>
</tr>
</tbody>
</table>

Conclusions

1. Software reuse enabled RNTDS programs to be produced and delivered at 12 to 21 percent of the cost of "writing from scratch," which equates to a five to eight-fold productivity increase over no reuse.

2. The relative cost, \(C\), is lower bounded by \(\frac{E}{n} + b\) for this architecture or 0.114. This equates to the case when \(R = 1\), or there is 100% reuse.

4.3. Gaffney Reuse Model

J. Gaffney provides extends the Gaffney/Durek model by providing some additional analytical techniques which are described in [Gaffney89].

The Relationship of Unit Cost Parameters to Gaffney/Durek

Let

\[
C_{US} = \text{Average cost per unit of the total software product}
\]

\[
C_{UN} = \text{Average cost per unit of new software in the product, and}
\]
\[ C_{UR} = \text{Average cost per unit of reused software in the product} \]

\[ R \text{ is still the proportion of reused code in the product} \]

Then,

\[ C_{US} = C_{UN} - (C_{UN} - C_{UR}) \times R \]

4.3.1. RNTDS in the Light of Gaffney Model

Gaffney notes that equation \(*\) is in the form of \( f(x,y) = mx + y \). The average values for \( C_{UN} \) and \( C_{UR} \) may be obtained from a linear fit of \( C_{US} \) versus \( R \), for fourteen projects within the architecture. Such a plot is shown below as Figure 3.

![Cost per Unit vs % Reuse (R)](image)

Figure 3 - Derivation of \( C_{UN} \) and \( C_{UR} \) for RNTDS Architecture

The y-intercept provides the average value of \( C_{UN} \), which is 11,282.78. The slope of the resultant linear fit is -10,805.10, which indicates that an average cost per unit of reused software is 477.68.

The value \( \frac{C_{UR}}{C_{UN}} \) reveals that for this architecture reusing a unit of code is approximately \( \frac{477.68}{11,282.78} \) or 4.23\% of the effort required to code that unit from scratch.

The following table provides values of \( \frac{C_{US}}{C_{UN}} \) for each project. This value represents the unit cost for each project relative to the cost if no reuse were used. This number is directly comparable to the relative cost, \( C \), computed for each project using the Gaffney/Durek model in section 4.2.1.
Table 4: Relative Project Cost, \( \frac{C_{US}}{C_{UN}} = 1 - \frac{C_{UL} - C_{UR}}{C_{UN}} \times R \)

<table>
<thead>
<tr>
<th>Project</th>
<th>R (% reused code)</th>
<th>Relative cost: ( \frac{C_{US}}{C_{UN}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0%</td>
<td>1.00</td>
</tr>
<tr>
<td>B</td>
<td>95%</td>
<td>0.09</td>
</tr>
<tr>
<td>C</td>
<td>89%</td>
<td>0.15</td>
</tr>
<tr>
<td>D</td>
<td>97%</td>
<td>0.07</td>
</tr>
<tr>
<td>E</td>
<td>99%</td>
<td>0.05</td>
</tr>
<tr>
<td>F</td>
<td>99%</td>
<td>0.05</td>
</tr>
<tr>
<td>G</td>
<td>99%</td>
<td>0.05</td>
</tr>
<tr>
<td>H</td>
<td>99%</td>
<td>0.05</td>
</tr>
<tr>
<td>I</td>
<td>89%</td>
<td>0.15</td>
</tr>
<tr>
<td>J</td>
<td>93%</td>
<td>0.11</td>
</tr>
<tr>
<td>K</td>
<td>99%</td>
<td>0.05</td>
</tr>
<tr>
<td>L</td>
<td>99%</td>
<td>0.05</td>
</tr>
<tr>
<td>M</td>
<td>99%</td>
<td>0.05</td>
</tr>
<tr>
<td>N</td>
<td>99%</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Conclusions

1. Software reuse enabled RNTDS programs to be produced and delivered at 5 to 15 percent of the cost of "writing from scratch," which equates to a 7 to 20-fold productivity increase over no reuse.

2. The relative unit cost, \( \frac{C_{US}}{C_{UN}} \), is lower bounded for the architecture by \( \frac{C_{UR}}{C_{UN}} \) or 4.23%. This equates to the case where \( R = 1 \) (100% reuse).

4.3.2. Comments on Gaffney/Durek Models

The Gaffney/Durek and Gaffney reuse economic models properly demonstrate that software reuse is a significant enabling technology toward improving software development productivity.

The benefits of reuse increase as more phases of software development are impacted. This effect is captured by the Gaffney/Durek parameter \( b \).

A reusable component is not more expensive to produce than a "non-reusable" one. The increased unit cost due to the creation of a reuse repository and software support environment is offset in the RNTDS architecture by a decreased testing requirement. This effect is captured by the Gaffney/Durek parameter \( E \).
Economic models of software reuse attempt to answer

- How do we decide when the benefits of reuse outweigh the cost?
- How do we estimate the cost of making a component reusable?
- How do we estimate the benefits of reusing a component?

The economic models are very useful frameworks for general discussion. If the models are to be used to predict specific answers to some of the above questions before the project starts, then more reports of project reuse experience, in terms of these models, are needed. Although general guidelines are suggested by the models, a project contemplating a reuse investment will find difficulty in predicting its own $E$ and $b$ parameters without reports of experience from similar efforts. With a broad, empirical knowledge base, economic models of software reuse can become a useful project planning tool much in the manner that software cost models are used today.

The models provide, as reported in Tables 3 and 4, comparisons of project performance against no reuse. A more meaningful contrast would be an economic comparison among various levels of reuse. Such a hypothetical comparison, within the framework of the models, is a potential area for further study.
5. CONCLUSIONS

1. Software reuse, as implemented by the RNTDS architecture, enabled Combat Direction System computer programs to be produced as low as 5 (by Gaffney model) to 12 (by Gaffney/Durek model) percent of the cost to produce them without reuse. This translates to a 8 to 20-fold productivity increase.

2. The organizational characteristics cited by the Software Engineering Institute (Kang and Levy) as promoting reuse hold true for the RNTDS architecture.

3. The high degree of reuse in the RNTDS architecture (89 to 99 percent across fourteen distinct programs) was made possible by the high degree of common processing functions required by the fourteen programs and from the use of design techniques, such as the uses of abstraction and small Ada-like components. Consequently, projects which foresee similar functional commonality and organizational characteristics may see similar benefits from reuse.

4. The benefits of reuse increase as more phases of software development are impacted. This effect is captured by the Gaffney/Durek parameter $b$.

5. A reusable component is not more expensive to produce than a "non-reusable" one. The increased unit cost due to the creation of a reuse repository and software support environment is offset in the RNTDS architecture by a decreased testing requirement. This effect is captured by the Gaffney/Durek parameter $E$.

6. The Gaffney/Durek and Gaffney reuse economic models properly demonstrate that software reuse is a significant enabling technology toward improving software development productivity.

7. More empirical information, specifically project reuse experiences reported in terms of the models, is needed to make the models more useful for projects which are still in the planning stages.

8. The models provide, as reported in Tables 3 and 4, comparisons of project performance against no reuse. A more meaningful contrast would be an economic comparison among various levels of reuse. Such a hypothetical comparison, within the framework of the models, is a potential area for further study.
ANNOTATED BIBLIOGRAPHY


The REusing Software Efficiently (REUSE) system "represents an initial attempt at solving the problem of cataloging and retrieving reusable software information." It contains a front-end information retrieval system which supports menu-driven and keyword searches, and the addition, change and deletion of components. Components in the library are classified as: template, module, package or program.


Survey of some progress and inhibitors. Different levels of reuse are: program statements, procedures/functions, subsystems and systems. (From abstract.)


Article proposes that the reuse process should begin with (1) identifying actually or potentially reused components, then (2) decomposing or abstracting those components into a generic form for cataloging and later use. Cites Draco project.


While not taking on the reuse problem directly, this well-known text does address some related issues and techniques, specifically: object oriented design, programming in the large and Ada programming support environments. Text successfully presents Ada on two levels: (1) a tutorial on the language; and (2) qualitative impacts of Ada and associated techniques upon the software life cycle.

[Booch87] Booch, Grady, Software Components With Ada, Benjamin/Cummings, Menlo Park, CA, 1987. More appropriately titled: "Reusable Software Objects With Ada," Text is designed to "train the reader in the creation and application of (reusable software) components." Begins with basic reuse and object oriented concepts, implements many different data structures, provides some useful tools and utilities, and discusses concepts pertinent to system construction from reusable components. Brief but pointed discussion of managerial issues.


Addresses acquisition, technical and managerial issues which inhibit widespread software reuse.


Short (4 page) paper which announces an Ada repository containing over 40Mbytes of source code, documentation and information files. This small repository claims a user community of 65 military organizations, government labs, corporations and colleges. Paper provides details for accessing the repository and how to get further information.

[Deutsch83] Deutsch, L. Peter, "Reusability in the Smalltalk-80 Programming System," Tutorial: Software Reusability, IEEE, Computer Society Press, Washington, DC, 1983. Smalltalk-80 supports 3 different kinds of reuse: reuse across hardware environments, reuse of algorithms across different data types, and reuse of program modules in different applications. Appears that these important concepts have been incorporated in other languages.


Authored by task force appointed by Under Sec. of Defense (Acquisition). Claims DoD software engineering practices lag far behind the state of the art. Names some initiatives which have the potential to yield an order of magnitude improvement in software productivity, reliability and timeliness: Ada, STARS project, DARPA's Strategic Computing Initiative, Software Engineering Institute and a planned program within the Strategic Defense Initiative. The recommendations were centered on increasing high-level coordination and support for the above efforts. (Report appears in excerpted form which concentrates on Ada-related findings. Reuse is not mentioned directly although some of the projects cited are attempting to foster software
reuse.)


Article proposes a reuse library constructed of software components stored as abstract data types (ADTs). An ADT has two parts: a descriptor, which stores knowledge about the component; and one or more implementations. Takes an object oriented view of design (which makes it easier to view programs as collections of data structures.) Provides a detailed description of the data stored in the descriptors. Describes library utilities required for component selection, retrieval and program building. ThorOUGH, detailed description of proposed reuse library.


Unclassified Navy manual which describes the control and production of Combat Direction Systems (CDS) programs under the Restructured Naval Tactical Data Systems (RNTDS) architecture. It describes the retrieval, addition and maintenance of reusable software components in the RNTDS Common Reusable Library, the transformation of program requirements into new library components and the status and control mechanisms used in the software development process. Mailing address: Commanding Officer, FCDSSA Dam Neck, Virginia Beach, VA 23461-5300.


Tools which support design should assist not only in designing new software components, but in reusing existing ones. Describes Wlisp, a knowledge-based object oriented tool and library. Detailed discussion of Wlisp user interface with little information on how components are stored.


Establishes a broad hierarchy of reusable information: from code fragments, logical structures and functional architectures to external and environment-related information. Reuse applies to all software engineering products. Proposes a five question sequence to be followed when reuse is contemplated. Surveys existing research and categorizes it in terms of his information hierarchy model – model appears to work well for this. Model implies that reuse can be pursued at several levels.


Lead essay of tutorial. Broadens "reuse" to mean "reusable software engineering" (which implies that any product of the software development process may be reused.) History: Mcilroy at 1969 NATO Conf on S/W Eng sounded "first call" for reuse. Since then first real efforts have been libraries of standard functions (i.e. math). City Raytheon, Toshiba and DoD (STARS/Ada) as leading efforts in 1970's. ITT-sponsored workshop in 1983 assisted broad, coordinated efforts. Today (1987): Have just scratched the surface. Economics and product complexity are prime motivators. Tools are needed.


Presents three economic models of software reuse which demonstrate potential cost savings. No actual project data is provided. Additional related models are presented which demonstrate the potential impact on quality and schedule.


Provides some extensions to [Gaffney88], particularly with respect to analyzing project data, amortization and optimum investment estimation.


Article examines current procurement, commercial and software engineering practices which inhibit widespread implementation of reuse.
Common communications and built-in test requirements for a mission processor were abstracted into common modules. Result was: common interfaces, reduced software development effort and time, reduced code size, and reduced test requirements. (From abstract.)

While Module Interconnection Languages (MILs) provide a "formal grammar construct to decide the various module interconnection specifications required to assemble a complete software system" [Prieto-Diaz86], MILs attempt to describe what each software component is supposed to do in terms of its interior design. The proposed syntax is Ada-like, but is extended to allow the association of several package bodies to a single package specification, the use of axiomatic specification and explicit interconnection commands for specifying how to build a system from its components.

Paper updates methods presented by Parnas based on lessons learned from a recent project which applied Parnas' SCR techniques. Presents a modified life cycle model, a more stringent linking of various software development documents and emphasizes use of reviews. Techniques addressed support reuse by emphasizing information hiding.

Article suggests means to enhance reusability of code and surveys some reuse measurement techniques. (From abstract.)

SEI's eighty-six entry bibliography on domain analysis and software reuse issues. Chronological, topical and author cross-indexing is provided.

Software Crisis motivation: Claims reuse has potential to improve software productivity "by an order of magnitude or more." Improvement in only one phase of development will not have significant overall impact; all phases of development need to be addressed: development environments, cheaper hardware resources, language improvements, transforming prototypes to products, off-the-shelf software, applications generators, formal specification and transformation systems, and very high level program generators/domain analysis. Examines Draco and SAFE systems. Good (and truly expansive) survey of techniques. Economic prediction seems reasonable if broad reuse occurs as described.

Motivation: Lots of software exists and a high degree of commonality exists in all domains which can be exploited. "Probably less than 15% (of new software) is unique, novel and specific." Remaining 85% is his target for reuse, which can occur at any of five hierarchical levels: data, designs, architecture, modules and programs/systems. Few software subfields have design standards. As an example of product overkill he discloses that 500 separate accounting programs, 300 payroll programs and 200 mailing list programs are currently commercially available. Claims that reuse of previously validated components can increase programming productivity by 60 to 80 percent.

From Software Engineering Institute. The selection of a software development methodology should include economic considerations. Compares some existing development methods. Presents some economic models, including one which provides a qualitative analysis of conditions needed for software reuse.


This report details one method for performing a domain analysis which focuses on the identification of distinctive features of the software systems in a domain. A comprehensive example of FODA application is provided.


This implemented system facilitates reuse of software components by storing them as generic schema which can be instantiated to handle different data types. An algebraic (pre- and post-condition) specification expressed in first-order logic is stored with each component. Good description of transforming components from generic to specific. Key-word search is used for retrieval.


This article outlines some of the attributes of Unix which support software reuse, including: function libraries, the pipe mechanism, hardware abstraction and program combination through input/output redirection.


Despite expansive-sounding title, after studying 5000 Cobol programs the authors simply conclude that "we do basically the same kinds of programs year in and year out." Claims potential for 50% productivity gain with little supporting evidence.


A unique paper which describes the "growth" of reusable objects from the "seeds" of a data dictionary. Uses EBNF-type notation. Appears to support reuse by supporting program evolution and modification. (Entertaining metaphor of describing code growth in horticultural terms is carried throughout the paper.)


Article asserts that in order to rigorously specify the allowable uses of each software component in a library, two separate formalisms are required: an "intramodule view" which describes the interior design of a component; and an "intramodule view" which expresses the relationships of components and their behavior only in terms of exterior (or interface) specification. Sounds similar to the later concept of LILs and MILs [GOGUEN86], [PRIETO-DIAZ86]. Proposed syntax extends Ada to include facilities for algebraic specification.


Claims that reuse of a component is 1/20th the effort required to build from scratch -- based on some unspecified project experience. With 80% reuse, 76 to 84% effort savings are achievable. Supports object oriented approaches and "Objective C" (a C extension).

Fascinating article by Toshiba Corp manager. Covers wide range of issues relating to the establishment of a “software factory” including such issues as capability assessment and careers, quality, project management and reuse. Reuse is seen as essential to project success. Describes reuse-oriented production organization, including the presence of a “software item” database accessed by key-word searches. Surprisingly, not much here which would be considered high-tech.

Examines impact of reuse on design phase and suggests how to exploit reusability in hierarchical specifications. (From abstract.)

Draco attempts to assist in the development of similar systems by capturing the results of analysis and design in a way which facilitates reuse. The mechanism uses different languages to semi-automatically transform specification to code. Motivation for emphasizing design reuse over code reuse is increasing costs of errors which occur in earlier phases of development. The challenge is constructing a syntax and structures which represent the desired domain. An example in the domain language of augmented transition networks is provided. This article is the first of five in this volume which describe experiments in achieving reuse through program transformation.

Fourteen presentations which provide a sample of current industrial reuse and reengineering state-of-the-art. Sponsored by the National Institute for Software Quality and Productivity, P.O. Box 70555, Washington D.C.

Twenty-three government and industrial presentations on reengineering processes, language-specific research and reengineering methods and tools. Mailing address: Officer-in-Charge, Naval Surface Weapons Center, 10901 Hampshire Ave, Silver Spring, MD 20903-5300.

The fact that we generally consider abstraction or information hiding to be essential for controlling software costs and for reuse of components comes from seminal articles such as this one. A concise overview of Parnas’ classic 3-module design (hardware, behavior and decisions) of A-7E avionics software. Provides a general schema for module design. If written today, the article would probably go into more detail about how using many small modules can support this overall schema.

This presentation in [NISQP91] by the Conte! Technology Center provides a high level discussion of software cost and reuse modeling. It borrows the reuse producer/customer paradigm from [Kang89] and uses it to call for a cost amortization scheme within “cost sharing domains.”

Module Interconnection Languages (MILs) attempt to assist programming-in-the-large by providing formal grammar constructs for describing the overall structure of a complex program and for specifying module interconnections. Automatic processing may be employed to verify system integrity. This expansive and detailed paper surveys the MIL state of the art: basic concepts, four MILs and some implementations.

Code reuse involves 3 steps: (1) accessing the code, (2) understanding it, and (3) adapting it. One way to reduce the effort of
these 3 steps is by carefully classifying the components. Paper proposes a "faceted" classification scheme akin to the Dewey decimal system in library science, except that syntactical relationships are used to resolve classification ambiguities (such as: Does the title "Nutritional Requirements of the Grizzly Bear" get filed under "Bear - Nutrition" or "Nutrition - Bear"?) Paper covers many issues, including: vocabulary control, depicting conceptual closeness of components, metrics for evaluating reusability of components, defining a query method and measuring the effectiveness of the classification scheme (which the authors claim is at least an improvement over keyword searches.) Excellent treatment of classification theory and implementation issues.


Consortium of 14 aerospace firms sees reuse challenge as being lack of standard terminology, standard concepts, tools and cost/benefit model. Canonical design - a technique based on families of similar components. (See [Parnas76] for seminal article.) Domain Analysis - a technique of breaking down an application domain into standard components, which become a standard for organizing a reuse library for that application. SPC Reuse program has 4 projects: a reuse library, a canonical design/synthesis effort, methodology study (how to better write reusable code), and a measurement project for quantifying effects of reuse.


From Software Productivity Consortium, a coalition of 14 U.S. aerospace companies with the goal of increasing software productivity and suitability. This article surveys SPC's "reuse activities and approaches to describing and enacting reuse processes." (From abstract.)


The PROTRAN system, in which run-time semantic checks of intermodule information is made, is described. PROTRAN attempts to attack the problem of specifying interfaces among reusable parts. Many of the features described appear to have been implemented by Ada's use of strong typing, rather than by translating data types as they are passed as this paper suggests.


Excellent survey article which categorizes and summarizes software reuse efforts.


Since we are reusing software more widely, it would be more beneficial to ensure that "good" design decisions are made. Information regarding what the component has been optimized for (if anything) should be stored with the component. (From abstract.)


Article uses an object oriented approach to produce a formal description of a domain (example domain is a fruit juice processing plant.) Apparently is a useful technique, particularly for narrow, complicated domains where many similar systems may exist.


Article and project attempt to support reuse by supporting the adaptation of existing programs through a process of reverse engineering, modularization, adaptation and code generation. Tool set was demonstrated by converting a somewhat unstructured COBOL-74 program and generating an equivalent (they claim) modular COBOL-85 program.


Expansive survey of most software engineering topics which describes the techniques and issues which relate to each phase of software development. Concise listing of motivators and inhibitors to reuse; then limits discussion of techniques to abstract
components (a la Booch). In object oriented design chapter, the author provides examples where object oriented design may be less useful than a functional design.

STARS (Software Technology for Adaptable Reliable Systems) is a Dept of Defense sponsored project with a goal of "supporting modern reuse-oriented software processes with open architecture software engineering environments (SEEs)." After completing an exploratory phase in April 1990, the project is now turning its efforts toward "fostering commercially available solutions to the DoD problems." Current activities of the three prime contractors are:

Boeing:
- Software Engineering Environment specification
- Technology exploration (Object Mgmt Systems [OMSs], X Windows and Andrew File System
- Object Manager Technology - evaluate a prototype OMS

IBM:
- Conduct a SEE domain analysis
- Integration of some commercial SEEs
- Reuse library

Unisys:
- Methods/tools for formal proof methods/logical verification
- User interface standardization
- A domain-specific environment and repository.

This, the second of the project's quarterly newsletters, summarizes various STARS project activities. Email address for the newsletter publishers is: newsletter@stars.rosslyn.unisys.com.


Describes implementation of a library of objects and a tool facilitating their reuse. Highly structured query language is used. Graphical user interface facilitates integrating the components.


Clear concise article concludes with realistic economic motivation. Motivators to reuse: Productivity (less new code is required) and Quality (use of previously tested code). Sees these effects as synergistic. Inhibitors to reuse: People. Programmers ("not invented here"/"not as good as mine" attitudes); Managers (Few tools exist, have to trust code which wasn’t produced here); and Computer Scientists (Other productivity approaches exist: Very high level languages, applications generators and executable specifications through transforms -- all of which compete for attention.)


Reuse of components and resources as a precursor to Ada. Various Ada properties support reuse. Object oriented programming in Ada and Lisp. Applications generators contribute in some domains. Knowledge engineering and applicability of reuse in AI domain. Very philosophical treatment of reuse topics.


"The requirements for a maintenance support system are similar in many respects to those for a reusability support system." In fact, these authors (and others) contend that "software maintenance is a particularly intense form of reusability." As a step toward the development of a "Software Maintenance Support Environment (SMSE), this report identifies the issues which a maintenance methodology must address and to identify the pertinent technologies. A maintenance paradigm of (1) understanding the existing system, (2) analysis and design of required changes, and (3) implementation is outlined. A conceptual design of an SMSE is presented with component parts explained.

Programming a small (50 line) module or task is one type of effort. Constructing a complex program from new and reused modules is entirely another. This second effort is termed "programming-in-the-large." Some software development concerns for programming-in-the-large are: life cycle support (modifying components), precise interface control (can communication occur between two components which are not using the same data types?), traceability, version control, et.al.
Other References Regarding Software Reuse


