ABSTRACTION-BASED REUSE REPOSITORIES

Grady H. Campbell, Jr.

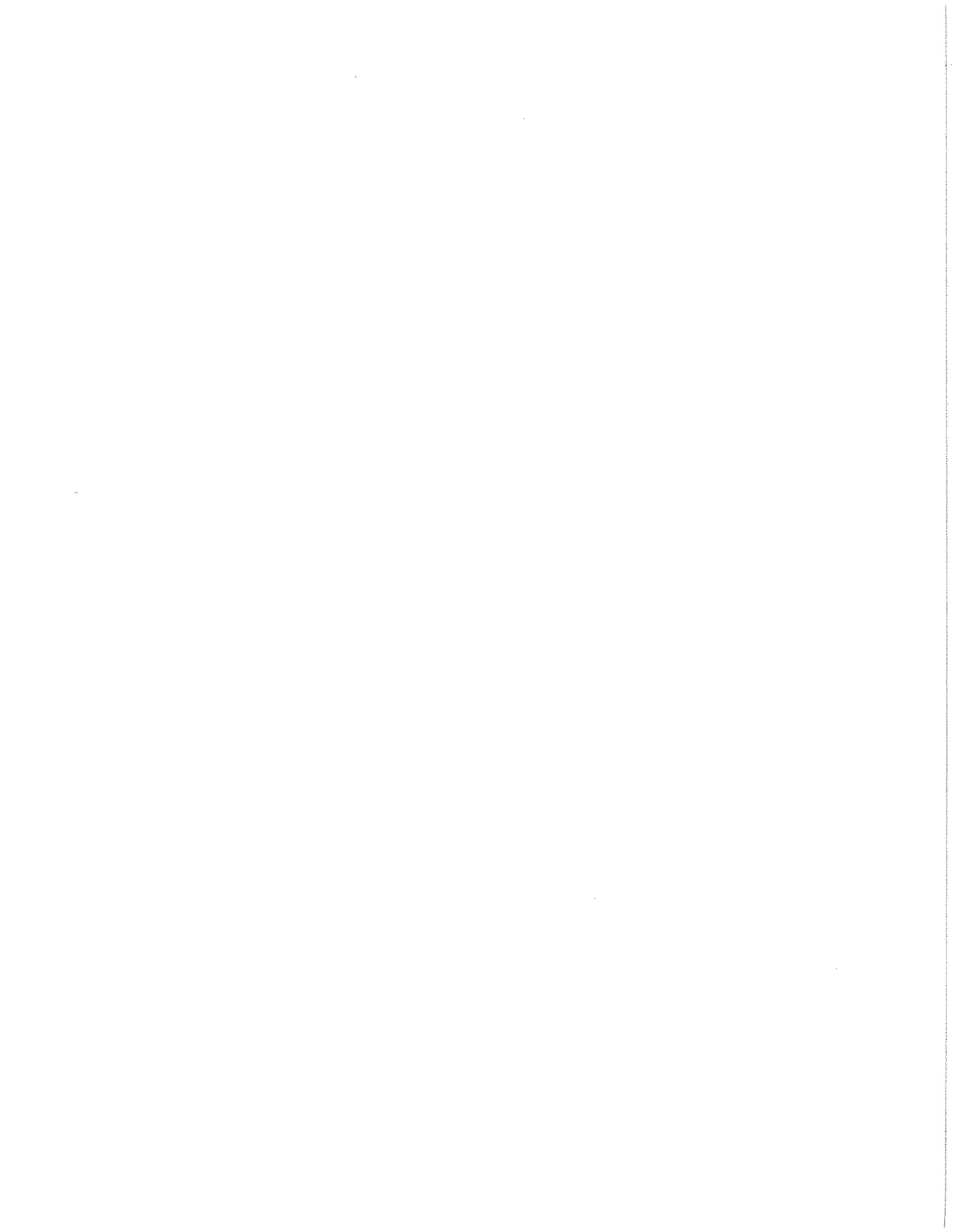
REUSE_REPOSITORIES-89041-N

VERSION 1.0

JULY 1989

SPC BUILDING
2214 ROCK HILL ROAD
HERNDON, VIRGINIA 22070

©1989 SOFTWARE PRODUCTIVITY CONSORTIUM, INC.
ALL RIGHTS RESERVED



ABSTRACTION-BASED REUSE REPOSITORIES

REUSE_REPOSITORIES-89041-N

VERSION 1.0

JULY 1989

This paper is partially based on ideas developed while the author was with Software Architecture & Engineering, Inc., of Arlington, Virginia.

This document may be distributed without restriction.
All complete or partial copies of this document must contain a copy of this page.

STATUS:		APPROVALS		
Working Draft	☐	Author:	Grady H. Campbell, Jr.	7/31/89
For Review	☐	V.P.:	Arthur Pyster	8/1/89
Final	☒	C.E.O.:	Judson B. Neale, Jr.	8/1/89

This paper will also be presented at the "AIAA Computers in Aerospace VII Conference," Monterey, California, October 1989.

SOFTWARE PRODUCTIVITY CONSORTIUM
SPC BUILDING
2214 ROCK HILL ROAD
HERNDON, VIRGINIA 22070

© 1989 SOFTWARE PRODUCTIVITY CONSORTIUM, INC.

ABSTRACT

A conventional view of a repository for software reuse is as a database of software components. Reusing a component from such a repository requires extracting candidates through a search of the database, understanding the candidates and selecting the best match, and adapting the selection to suit the intended use. These activities are time consuming, none are guaranteed to produce an acceptable result, and improvements in one can increase the cost of the others. As an alternative, the concept of an abstraction-based repository is proposed. An abstraction-based repository is conceived as a taxonomy of abstractions where each abstraction is a characterization of a family of software components. Reusable components are obtained by selecting and instantiating a family abstraction guided by resolution of a set of prescribed design decisions.

TABLE OF CONTENTS

Section | Page

1. **Introduction** **1-1**
   - 1.1 Overview 1-1
2. **A Current Model of Reuse Repositories** **2-1**
   - 2.1 Assumptions of the Model 2-1
   - 2.2 Problems with the Model 2-2
3. **Conceptual Foundations of Abstraction-Based Reuse** **3-1**
4. **Abstraction-based Reuse Repositories** **4-1**
5. **Supporting Abstraction-Based Reuse** **5-1**
   - 5.1 Metaprogramming 5-1
6. **Generalizing Existing Software for Reuse** **6-1**
7. **Constructing Systems From Components** **7-1**
8. **Related Work** **8-1**
9. **Future Direction** **9-1**
10. **References** **10-1**

LIST OF FIGURES

Figure No. Page

Figure 2–1: Conventional Reuse Repository Model 2–1

Figure 4–1: Proposed Reuse Repository Model 4–2

Figure 4–2: Instantiations of a Family P 4–2

Figure 5–1: Constructs of a Metaprogramming Notation 5–2

Figure 5–2: Sample Family/Instances Description 5–2

1. Introduction

Software reuse is intuitively attractive as a means to increase software productivity and reliability. Unfortunately, attaining this promise does not seem imminent[3]. The effort required to find and adapt components for new uses seems excessive. This paper attempts to relate the causes for this to the prevalent model of reuse repositories and suggests an alternative model that seems to result in effective support for significant reuse when the likely nature of future uses can be anticipated.

For the purposes of this paper, a software system is viewed to be a composition of assemblies (subsystems). An assembly is an integrated collection of parts. A part corresponds to an information hiding module[12] and includes an abstract interface, implementation, test, and documentation facets.

1.1 Overview

Conventionally, a reuse repository has been thought of as a database of software components that are to be used either directly or with minor modification. Such an approach, while making significant software reuse possible, fails to make such reuse practical. This failure is attributable to the conflicting needs of the extraction, selection, and adaptation activities.

An abstraction–based reuse repository addresses these difficulties with a unified solution. This solution derives from the observation that a set of software components that satisfy similar needs constitute a software component family. A significant aspect of reusing a component conventionally is the modification of the component in minor ways to suit particular needs (i.e., the transformation of one family member into another). If, instead of starting with individual family members (i.e., component instances), it is possible to start with an abstract description of the entire family, then all possible family members could be derived directly without this transformation between members. Such a derivation is in fact possible given abstract descriptions in the form of 'metaprograms' that explicitly distinguish shared traits of family members from differentiating traits. From this basis, a view of reuse repositories results in which abstractions populate a taxonomy of constructable components and component instances are delivered automatically as instantiations of abstract component families.

2. A Current Model of Reuse Repositories

The conventional model for software reuse is based on the assumption that it is more productive to take an existing component which is a close–fit and modify it for reuse than it is to build from scratch. Under this model, utilizing software from a reuse repository (Figure 2–1) depends on three activities: extraction, selection, and adaptation. Extraction is the process by which the contents of the repository are rapidly searched to find a set of candidate software components such that each is suitable to some degree for the intended use. Selection is the process by which the applicability of each candidate component is analysed in detail and evaluated so that the most suitable component may be reused. Adaptation is the process by which a selected component is reliably modified to suit the intended use exactly.

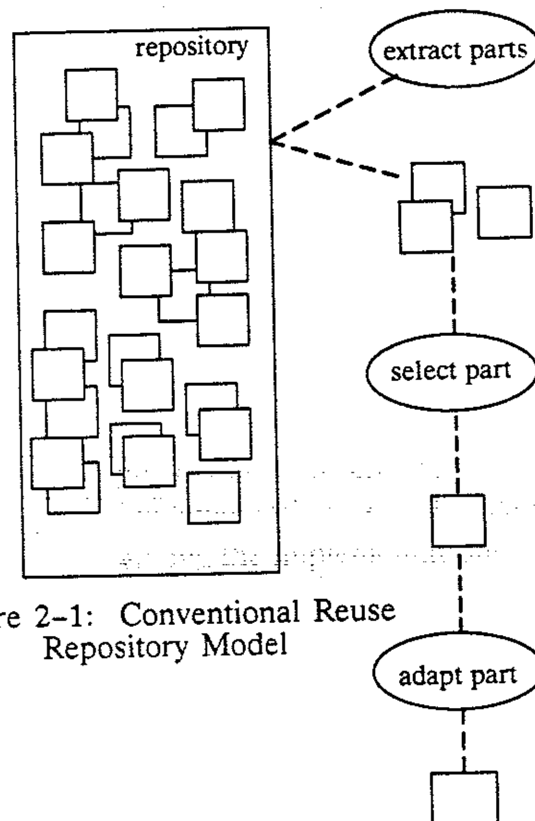


Figure 2–1: Conventional Reuse Repository Model

2.1 Assumptions of the Model

There are several important assumptions underlying this model:

- reusable components are explicitly stored as discrete entities analogous to (a set of) files
- there exist a classification scheme and a set of attributes that characterize any component such that the extraction decision does not require detailed scrutiny of components

- the number of components identified in the extraction process will be small enough to allow a final selection to be based on detailed analyses of candidate components
- there will be enough variety among the available components so that the effort for adaptation will be significantly less than the effort for implementation from scratch

2.2 Problems with the Model

Assuming that only explicitly stored components will be available from a repository is impractical and inefficient given the reality that a useful repository must make an enormous number of components and component variations available. This assumption is fundamental to the opportunistic (scavenging) model of repository populating wherein components are taken as–is from existing programs. To keep the cost of adaptation significantly below that of creation from scratch, the repository must provide sets of functionally similar components that vary in well–defined ways corresponding to the actual, future needs of users. Without such variety, the cost of adaptation will easily become unacceptable and therefore fail to make reuse sufficiently economical. These two assumptions together result in the need for an extremely large number of components that would be costly in storage costs and require a complex classification scheme and attribute set to make extraction manageable.

On the other hand, a large repository of similar components could make the selection process too costly; extraction must produce a very small number of candidate components, preferably one, for the selection activity to be feasible. While restricting the application domain of a repository could reduce the necessary storage space and reduce the cost of the extraction activity, it cannot reduce the costs of selection or adaptation if either requires that a component be fully understood in detail.

While it is possible to imagine a sufficiently complex classification scheme and attribute set to support a large repository, there does not seem to be an adequate, existing categorization. Taking such a categorization for granted, classifying an arbitrary component would be difficult and error–prone. This suggests that reusable components should be designed initially to fit into such a categorization.

The problems of the conventional repository model lead us to the conclusion that an adequate repository must represent components in such a way that extraction produces only one (abstract) candidate representing a component family. A set of such components constitutes a family such that adaptation consists primarily of a set of prescribed decisions that result in the 'selection' (through an instantiation mechanism) of exactly the needed component. Such a repository will be referred to as an 'abstraction–based' reuse repository.

3. Conceptual Foundations of Abstraction–Based Reuse

Certain principles that are known to improve the quality of software have correspondences for effective reuse. When designing software, it is known that the cost of change can be reduced if anticipated changes in the design are hidden as secrets of information–hiding modules. Since all conceivable changes cannot be equally easy, the quality of the design is determined by the degree to which actual changes correspond to anticipated changes. Similarly, in reuse, the cost of reusing a component can be reduced if anticipated variations in its design are abstracted as decisions that the reuser can make to adapt the component to different needs. Much of the same discipline applies in design for reuse as in design for change. By making assumptions and design decisions explicit, there results a basis for making judgments about which conceivable variations are more or less likely across a family of components.

The concept that structuring information-hiding modules into a hierarchy helps a developer locate and understand modules[14] also applies to structuring of reuse repositories. In order to locate and understand a reusable component without undue effort, the reuser needs a conceptual model of what components are available and what their capabilities are.

The principle of separation of concerns applies in reuse to suggest the idea that the characteristics of a family be distinguished as common or differentiating. Common (stable) characteristics are subsumed by the abstraction of the family while differentiating (varying) characteristics are represented as parameters of variation on that abstraction. This is analogous to the creation of abstract interfaces to define the common features of a family of implementations and the identification of secrets that represent dimensions of variability among the implementations. Just as hidden decisions are easier to change, so are variable decisions.

Dijkstra[6] first articulated the view that program design should be concerned with the characteristics of a software family and not just a currently required program instance. This was a basis for the concept of stepwise refinement of programs. Parnas[13] extended this view to establish the concept of abstract module specifications as characterizing a family of implementations, each derived through a process of stepwise refinement. Both of these concepts of family were concerned primarily with how a program could be designed to facilitate anticipated changes in program requirements.

We go further to suggest that there is a concept of design families that can facilitate anticipated variations in requirements among programs that are conceptually similar and reasonably characterized as constituting a family of designs. This is not to suggest that the nature of variations among a family of programs is essentially different from that of likely changes as individual programs evolve; instead, this is an attempt to define a concept for describing component families in such a way that likely variation decisions can be made at the time of component reuse without requiring detailed selection and adaptation activities.

4. Abstraction–based Reuse Repositories

The abstraction–based model for reuse repositories is based on the premise that reuse will be productive only if there exists an integrating concept of component families from which specific components can be automatically derived. We go further to characterize a family of component designs that vary according to a set of decisions that are deferred until an instance of the family is needed, when a basis exists for resolving the decisions to meet specific needs. The decisions control the composition (using sequencing, conditionals, iteration, and nested instantiations) of fragments to form concrete instances that embody the decisions.

Abstraction is essential for practical reuse repositories. If only explicitly stored components are retrievable from a repository, there will be a large cost either for component storage space or for adaptation of each component before reuse. If implicitly stored components are retrievable, the storage space is reduced; if an implicit component definition is an abstract description of a component family, the adaptation cost is reduced as well.

In the proposed model of reuse (Figure 4–1), acquiring a component is reduced to two activities: family selection and instantiation. The retrieval of a component starts with the identification of a family. The set of families is organized into an information–hiding hierarchy which is traversed top–down to identify the appropriate component family that conceptually includes the required component. Associated with each family definition is a set of family–specific traits that provide differentiation criteria corresponding to a set of prescribed decisions (the sets of D_{xy} in Figure 4–2) whose resolution is sufficient to distinguish any two family members.

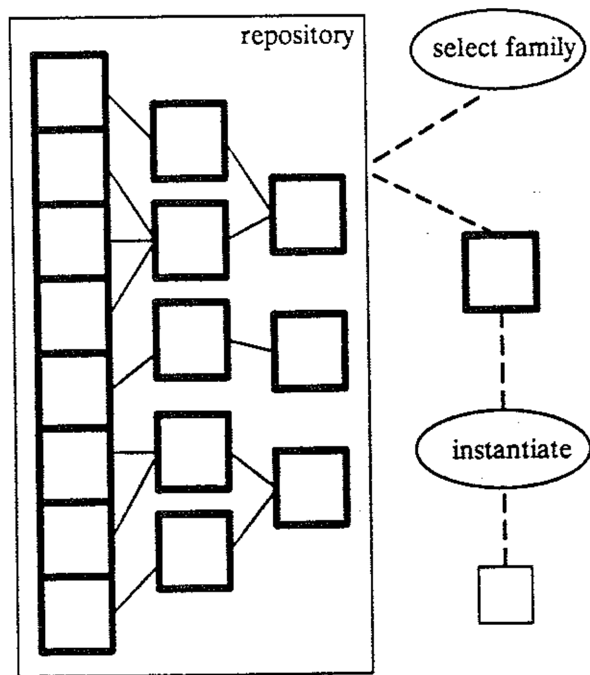


Figure 4–1: Proposed Reuse Repository Model

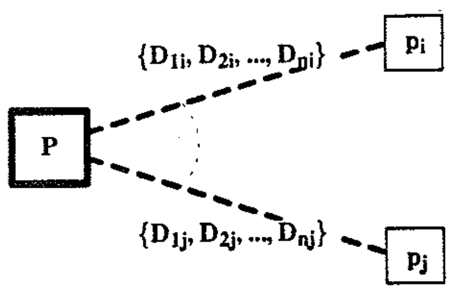


Figure 4–2: Instantiations of a Family P

5. Supporting Abstraction–Based Reuse

There are three elements necessary to the support of abstraction–based reuse:

- a taxonomy of reuse families (an information–hiding hierarchy);
- a notation for the composition of a family description out of component fragments;
- a mechanism for the instantiation of components from family descriptions.

The taxonomy provides the user with a conceptual model that organizes access to components in an understandable way. Without such a taxonomy of abstractions, users will not have an adequate sense of what components are available or how a design should be influenced to effectively exploit available components.

The notation for family description is a 'metaprogramming' notation. It provides a medium for programming the construction of components operating on component fragments and decision guidance (i.e., instantiation parameters).

The instantiation mechanism is a translator of the metaprogramming notation. It is the equivalent of a conventional programming language macroprocessor generalized and tailored to the domain of component definitions.

5.1 Metaprogramming

The representation of a component family is an abstraction of the representation of a software component. A family is represented through the parameterized composition of software component fragments. Metaprogramming is the construction and instantiation of family descriptions using this representation. Figure 5–1 identifies the major constructs of a metaprogramming notation. A family is described by a metaprogram definition; an instance of the family is any literal fragment that can result from the complete instantiation of that definition. Formal parameters may be literal fragments (for substitution through instantiation), sequences of fragments (for iterations), structures of fragments (for substitutions or control of alternations), or instantiations. Formal parameters may have associated constraints to prevent invalid instantiations.

Fragment composition is analogous to the process that software implementors generally follow in constructing a component. A component family is characterized as an abstraction of a set of instances that differ according to decisions sufficient to discriminate among the instances.

Figure 5–2 is a simple example of how a family and instances might be characterized abstractly. It is important to note that implementation decisions are not made explicitly by the reuser; particular decision combinations may result in very different component designs or implementations but this is determined by the reusable component's implementor.

Construct	Purpose
sequencing	a construction for concatenating patterns
alternation	a construction for choosing among alternate patterns
iteration	a construction for repeating a pattern
definition	creation of a named, parameterized pattern
instantiation	application of actual parameters to a pattern
literal fragment	a pattern containing no meta-constructs

Figure 5-1: Constructs of a Metaprogramming Notation

Family: ordered collection data types
Deferred Decisions:
(1) accessibility (FIFO, LIFO, indexed)
(2) element type
(3) capacity (infinite or bounded:integer)

Instance: infinite queue of integers
Resolved Decisions:
(1) FIFO
(2) integer
(3) infinite

Instance: stack of 50 'activation records'
Resolved Decisions:
(1) LIFO
(2) 'activation record'
(3) bounded:50

Figure 5-2: Sample Family/Instances Description

6. Generalizing Existing Software for Reuse

The metaprogramming process ideally supports design for reuse wherein anticipated variations among a family of components guide adaptation capabilities. This process can be applied as easily to the generalization of existing, proven software that can be reused if generalized after–the–fact to support needed variations. The key constraint is that assumptions and decisions that distinguish the component from other variations must be fully understood. This can be much harder starting with an existing component for which such variation was not anticipated at the beginning.

An analogous situation occurs when a family needs to be instantiated in a way that was not anticipated. Although the family could be instantiated for anticipated decisions and then adapted as in a conventional model of reuse, the abstraction–based model requires instead that the family definition be extended to accommodate other, future occurrences of the same variation. Since perfect anticipation of all conceivable variation is impossible, such situations cannot be avoided. The attempt, however, to anticipate even some variation leads to explicit statements of assumptions and decisions that make accommodation of new dimensions of variation less error–prone.

7. Constructing Systems From Components

With abstraction–based reuse, the construction of systems is largely the composition of component family instances. There are three methods of composition that are useful: uses–structure–based, instantiation–determined, and invocation–based.

The uses structure[14] for a component defines which other components are needed (from the reuse repository) for it to be complete. The implementor of a component definition both controls the composition and determines the characteristics of other, nested components, guided by the parameters to the defined component's instantiation.

The formal parameters of a metaprogramming definition may allow the substitution of particular component family instantiations in the instantiation of the definition. The composition is controlled by the implementor of the definition, but the reuser determines the characteristics of the substituted component supplied to complete the instantiation.

Invocation–based composition is the construction of a program that invokes interface facilities of component family instantiations. The reuser, in this case, controls both how nested components are composed and the characteristics of those components.

8. Related Work

There are several instances of formalization of aspects of abstraction–based reuse as advocated here. Goguen[8] and Dershowitz[5] give the same emphasis to the importance of abstraction. Lenz[10] describes a macro–processor–based, building block approach which is somewhat more limited but philosophically very similar to our approach. Bassett[2] and Polster[15] describe approaches that emphasize the composition of fragments, as in metaprogramming; the 'cliches' of Waters[17] are similar but emphasize generalization of instances through the introduction of 'role' placeholders. The fragment composition mechanisms of metaprogramming are similar in concept to the FP functional forms of Backus[1].

Transformational implementation approaches to automatic programming, such as Fickas[7], have some of the flavor of metaprogramming but tend to emphasize modelling of the process of program creation/composition rather than the modelling of the form and content of products to be created. Object–oriented languages support a generalization/specialization approach to program reuse, as described in Johnson[9].

Application generators[4] are similar to metaprogramming in that instances may be derived by tailoring and composing fragments. Levy[11] outlines an economic analysis that supports the use of an application–generator–packaged metaprogramming method.

Previous experience[18] using a metaprogram notation for family definitions, organized into an information–hiding hierarchy, and an associated instance generator as an instantiation mechanism demonstrated viability and effectiveness in facilitating software reuse. This use enabled the construction and use of a prototype, domain–independent application generation environment that showed significant (but unquantified) promise for increasing productivity and reliability.

9 Future Direction

The Software Productivity Consortium was established to seek significant improvements in software development productivity and product reliability. Software reuse has been identified by Pyster[16] as a key factor in the capabilities of an software development environment to achieve these goals. A part of this effort will be a further evaluation of the concept of component families and abstraction-based reuse.

10. References

1. John Backus, "Can programming be liberated from the von Neumann style?", *Communications of the ACM* 21, 8 (August 1978), 613–641.
2. Paul G. Bassett, "Frame-Based Software Engineering", *IEEE Software* 4, 4 (July 1987), 9–16.
3. Ted Biggerstaff and Charles Richter, "Reusability Framework, Assessment, and Directions", *IEEE Software* 4, 2 (March 1987), 41–49.
4. J. Craig Cleaveland, "Building Application Generators", *IEEE Software* 5, 4 (July 1988), 25–33.
5. Nachum Dershowitz, "Program Abstraction and Instantiation", *ACM Transactions on Programming Languages and Systems* 7, 3 (July 1985), 446–477.
6. E. W. Dijkstra, "Notes on Structured Programming" in *Structured Programming,* O. J. Dahl, E. W. Dijkstra., and C. A. R. Hoare, Eds., Academic Press, London, 1972.
7. Stephen F. Fickas, "Automating the Transformational Development of Software", *IEEE Transactions on Software Engineering* SE–11, 11 (November 1985), 1268–1277.
8. Joseph A. Goguen, "Parameterized Programming", *IEEE Transactions on Software Engineering* SE–10, 5 (September 1984), 528–543.
9. Ralph E. Johnson and Brian Foote, "Designing Reusable Classes", *Journal of Object–Oriented Programming* 1, 2 (June/July 1988), 22–35.
10. Manfred Lenz, Hans Albrecht Schmid, and Peter F. Wolf, "Software Reuse through Building Blocks", *IEEE Software* 4, 4 (July 1987), 34–42.
11. Leon S. Levy, "A Metaprogramming Method and Its Economic Justification", *IEEE Transactions on Software Engineering* SE–12, 2 (February 1986), 272–277.
12. David L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules", *Communications of the ACM* 15, 12 (December 1972), 1053–1058.
13. David L. Parnas, "On the Design and Development of Program Families", *IEEE Transactions on Software Engineering*, SE–2 (March 1976), 1–9.
14. David L. Parnas, Paul C. Clements, and David M. Weiss. "The Modular Structure of Complex Systems", *IEEE Transactions on Software Engineering*, SE–11, 3 (March 1985), 259–266.
15. Franz J. Polster, "Reuse of Software Through Generation of Partial Systems", *IEEE Transactions on Software Engineering* SE–12, 3 (March 1986), 402–416.
16. Arthur Pyster, "The Synthesis Process for Software Development", Software Productivity Consortium, November 1988.
17. Richard C. Waters, "The Programmer's Apprentice: A Session with KBEmacs", *IEEE Transactions on Software Engineering* SE–11, 11 (November 1985), 1296–1320.
18. Grady H. Campbell, Jr., "Abstraction–Based Environments", Software Architecture & Engineering, Inc., April 22, 1988.