TMM: Software Maintenance by Transformation

Guillermo Arango, Ira Baxter, Peter Freeman, and Christopher Pidgeon
University of California, Irvine

Porting an undocumented program without any source changes demonstrates the value of a transformational theory of maintenance. The theory is based on the reuse of knowledge.

Maintaining software with woefully out-of-date design documents—or none at all—is an unfortunate reality for many software practitioners. As needs change, software must be amended, or maintained, to adapt to the new environment. Often, such adaptation involves porting programs from one machine to another. If there is no information about original design decisions or abstractions, the software becomes obsolete, and the enormous resources invested in its construction are lost.

To avoid this waste, we propose a method that will allow practitioners to recover abstractions and design decisions that were made during implementation. This method, called the transformation-based maintenance model, or TMM,[1] is not just theory; it has enormous practical value as we proved when we successfully ported an undocumented, complex software system without *any* source changes.

We developed the TMM as a result of our own software maintenance frustrations. During our research on the reuse of software engineering artifacts and knowledge, we relied heavily on a prototype system, called Draco.[2-4] Because the computer on which the Draco system was developed and operated had to be removed, we were forced to consider porting the system to another operating environment. In other words, we had all the makings of a classic maintenance problem; namely,

- Other than technical reports and publications describing the conceptual ideas behind the system, the only documentation available was the source code.
- Members of the research team charged with responsibility for the port were inexperienced with the mechanisms used by Draco.
- The original author was not available for consultation.

The complexity of the task (and the desire to practice what we preach about reuse) compelled us to look for means of conversion other than manual reimplementation. Our problem was twofold: (1) we had to determine what the code was doing (leading to the idea of abstraction recovery), and (2) we had to reimplement it (here, we could use Draco, since it supports a paradigm for the implementation of software from components). Our porting effort proved successful.

General vs. domain-specific languages

After successful application of abstraction recovery, we began to develop the TMM as a formal model of software maintenance. The power of the TMM lies in the formal manipulation of problem domain and software design knowledge. Hague[5] discusses the idea of a "super" language customized for the application domain. Such a language would then map onto a real language. Hague claims that any gain in flexibility may be lost because the language may not compile on a real machine. Thus, he rejects the idea of a super language. The Draco paradigm suggests an alternative that makes compilation possible for domain-specific languages. We think the idea of a super language should be reconsidered—one for each domain.

Following an approach similar to Hague's, Boyle[6] employs an extended version of Fortran as a single base language. By extending the base, new classes of abstractions are expressed. In our paradigm, these qualify as separate domains and deserve separate notations. Boyle's approach suffers from the limitations impicit in wide-spectrum languages;[7] using a single base language limits the transformation alternatives to the primitives used by the language.

An earlier version of this article, entitled "Maintenance and Porting of Software by Design Recovery," was presented at the Conference on Software Maintenance, Washington, DC, 1985.

0740-7459/86/0500/0027$01.00 © 1986 IEEE

Managing change

Basically, we see four major reasons for change:

- Performance must be enhanced.
- The program must operate in a different environment.
- Different functions are required.
- A design error must be corrected.

Many authors[5,8,9] think these changes must have different solutions, as evidenced by their varying definitions of portability, transportability, and adaptability. We prefer to think of software maintenance as Boehm[10] does: "the process of modifying existing operational software while leaving its primary functions intact."

Because our method recovers design and uses it to reimplement the program, the reason for the change doesn't really matter. The reimplemented program may have different functions or enhanced performance; the TMM unifies the management of change regardless of its cause.

The Draco paradigm

The Draco paradigm is a method for constructing software systems from specifications. Each specification is composed of components from a mixture of domain areas relevant to the problem. The notion of a domain—an area of captured expertise—is fundamental. Within a domain is a set of components. The purpose of each component is to capture a semantic primitive for the domain. For example, the domain of arithmetic includes components for addition and multiplication. The domain of relational database includes components for select, project, and join. Each component of a specification is transformed into an implementation by selecting from a library of possible implementations.

The Draco paradigm is an instance of a more general class of domain-modeling approaches to software construction. (Partsch[11] gives an excellent overview of transformational systems for those unfamiliar with the transformational approach to programming.) The Draco paradigm assumes that one will construct a number of similar software programs. These programs share the property that they operate on objects from one (or usually more) domains. The domains of interest are formalized before program construction begins. This formalization effort, called domain analysis, addresses the difficult task of how to extract the common structures from a class of problems to produce the parts of a domain. Domain analysis is a key issue in the application of the Draco technology and one of the central research concerns of our project.

The description of a Draco domain consists of

- *domain semantics* expressed as an (informal) set of concepts (represented by *components*) composed of objects, operators, and relations;
- *domain language* described by a formal external notation for specifying a problem in the domain (a concession to human engineering);
- *an abstract graph schema,* which provides a formal internal representation for the notation in terms of the domain semantics;
- *domain parser,* a recognizer that maps sentences in the domain language onto the graph schema;
- *pretty printer,* a mechanism that generates domain language sentences from the graph schema (the inverse of the parsing process);
- *a set of transformations* that map internal representations in a domain to equivalent internal representations in the same domain, generally used for optimization; and
- *a set of refinements* that map individual abstract concepts in the domain to configurations of concepts in other domains closer to a target implementation.

Virtually all systems implemented with Draco define at least two domains: a high-level application domain and a low-level executable domain (e.g., Pascal, Lisp, assembly language). Software development using Draco starts with an abstract specification from a combination of domain languages. A component of a specification can be implemented using any applicable refinement; implementing a concept produces a revised specification closer to an executable form. The imple-

Why are software changes so difficult?

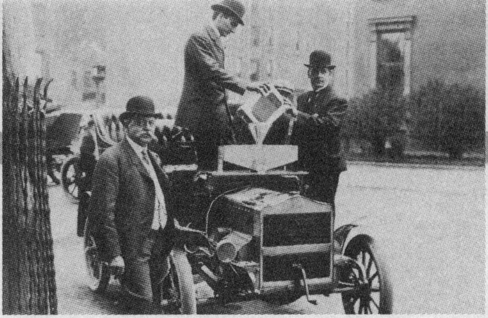
We spend a fortune on software design and development. As needs change, we would like to preserve our investments through appropriate changes in the products. It sounds simple enough, so why is the changing process so difficult?

In most cases, the original design is inaccessible. The original requirements analysis and specifications, if recorded, are out of date and do not correspond precisely to the code. Moreover, scattered throughout the code are idioms that correspond to idiosyncrasies in the current software environment. The code for even the most straightforward algorithms is almost always disguised by optimizations that depend heavily on both local context and on (inaccessible) global design decisions. Code for complex algorithms is nearly impossible to disentangle.

The parts of the design and environment that *are* documented are usually not machine processible. The maintenance task then becomes one of manual processing, which is both expensive and unreliable. There is also a popular perception that small changes in a system require correspondingly small efforts. Because we cannot understand the ramifications of a change, we optimistically assume that the change can be easily done. When inappropriately simple changes are made, they often introduce inconsistencies that later require extensive testing and further analysis to locate and debug.

Changes made to a program in the past leave scar tissue; code is not easily changed because of the ripple effect on the rest of the software. Over its lifetime, a system is dissected, modified, and sewn back together until its form is beyond recognition. Horror stories about these software Frankensteins are well-known to practitioners.

Our capacity to make changes to software systems is limited if we must rely on manual methods. At best, we can count on the maintenance team's ability, which can be enhanced only through better design. A more realistic choice is to transcend the limits on manual methods by using automation, as we have done with the TMM described here.



The Bettmann Archive

mentation process traverses a path through a space of possible implementations toward progressively lower abstractions until a concrete implementation is reached. The space forms an enormous directed acyclic graph, called a possible refinement DAG (Figure 1).

Nodes in the DAG represent specifications for the program written using the domain languages. The single root of the graph represents the initial system specification. Leaves are executable specifications, that is, programs. Intermediate nodes represent specifications at varying levels of abstraction. Each node except the root represents a correct partial implementation of the initial program specification. (Refinements are required to preserve correctness.) The higher the node in the DAG, the more abstract the specification. Arcs represent possible design choices (use of some refinement or optimizing transformation). An implementation for a specification corresponds to the path from the root node to one of the leaves. For the remainder of this article, the term "specification" can mean the original specification (root), an intermediate node, or an executable specification (leaf).

The refinement DAG is never constructed in its entirety. The only paths requiring exploration are those needed to reach the desired leaf from the root. Normally, only one path is explored; branches emanating from it represent rejected design choices.

For example, consider the DAG in Figure 1. The node n_2 could represent the specification containing the abstraction Sort-Symbol-Table; the arc r_2 might represent the design choice Use-Heapsort-Algorithm. The node n_3 would represent the refined specification, which includes the abstraction Sort-Symbol-Table-Using-Heapsort; that is, n_3 contains the code to implement Heapsort. Similarly, r_3 could represent the choice of data structure for the symbol table, and so on.

Usually, an individual node can be reached by many paths, representing differing orders of choice of the same set of design decisions. A path from the root to a leaf represents a particular choice of a set of implementation design decisions and constitutes what is generally called *the design*. This path can be viewed as an explicit representation of stepwise refinement as proposed by Wirth.[12]

A mechanism called *tactics* controls navigation through the graph. Tactics allows the application of the trade-offs used by the designer in decision making. Tactics can be used to achieve different implementation goals, such as speed, minimal space, and rapid prototyping, and separate tactics can coexist. Tactics may use data from a number of sources: the set of possible refinements, conditions on refinements, information stored in the tactics, the current specification, and even answers from a human. Tactics serves as the basis for a design's rationale.

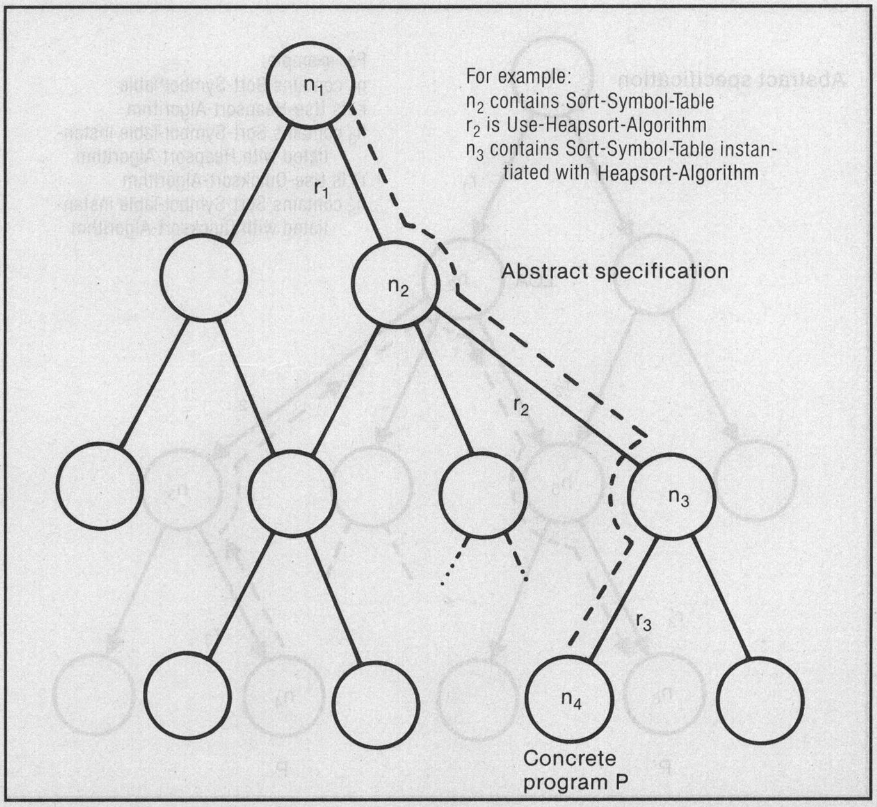
The Draco paradigm we have just described applies to all programs. We can see its applicability by considering a single refinement that converts a program specification into a particular implementation. While this view is somewhat trivial, it shows that all programs can be constructed from refinements; the only difference is how many.

Developing the TMM

In this discussion of our transformation-based maintenance model, we have made a number of assumptions. We assume that a program has been derived from a specification using the Draco paradigm. Further, we assume that the specification, the refinement DAG, and the implemented program are available to a would-be maintainer. (Later, we discuss how to perform maintenance when only the implemented program is available.) If many changes need to be made to a program, we make them one at a time.

Using the Draco paradigm, we can change a program in two ways:

(1) Choose a different implementation. Here, we choose an entirely new path through the refinement DAG from the ini-



For example:
n_2 contains Sort-Symbol-Table
r_2 is Use-Heapsort-Algorithm
n_3 contains Sort-Symbol-Table instantiated with Heapsort-Algorithm

Figure 1. Construction of program from specification refinement decisions r_1, r_2, and r_3.

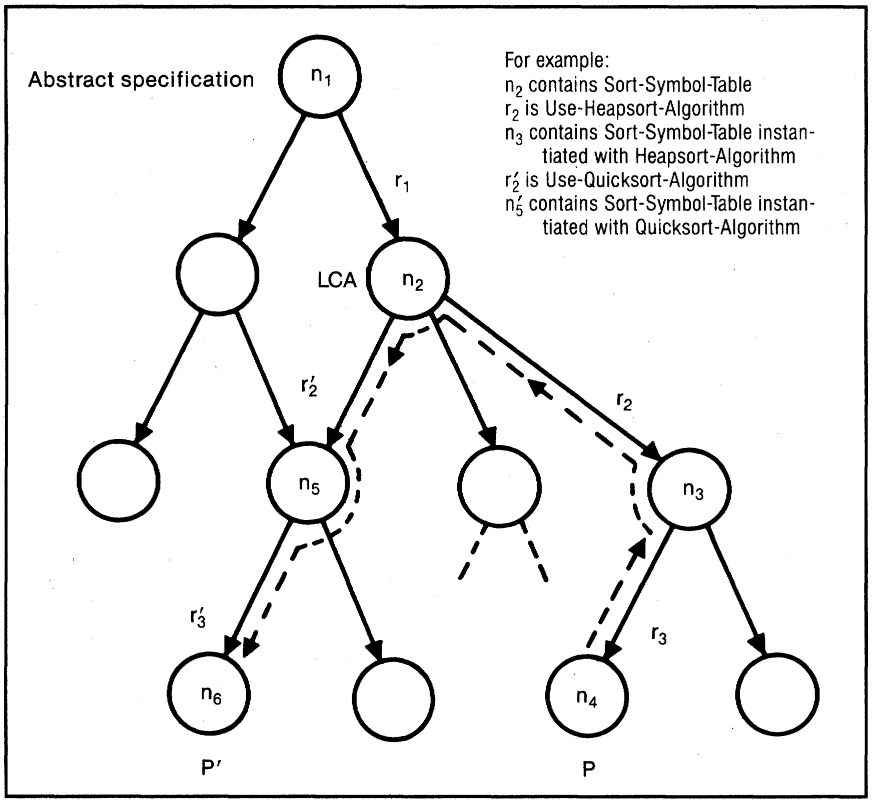
tial specification (root) to a different implementation (leaf).

(2) Start with the implementation and reverse some of the design decisions, moving up the refinement DAG toward the root. Having undone a sufficient number of design decisions, we then reimplement by making new choices—descending to a different leaf.

The first method is equivalent to reimplementing the program from scratch and is generally not preferred. Often, many of the design decisions made in the original implementation can be used again in reimplementation. For example, suppose the structure of the symbol table in a compiler must be changed. Some parts of the compiler coupled to the symbol table might also require change, but the implementation of other unrelated parts, such as the token builder, parser, and code generator, could remain unchanged. In other words, many of the original design decisions are reusable in this example.

The second method, in which some design decisions are reversed, involves a concept called the least common abstraction (LCA). At some point along the path up the refinement DAG toward the root, we reach a node that encompasses both the current (undesired) and desired implementations. This node is called the LCA. It is the top node of an embedded sub-DAG and can be reached by any of several paths in the original DAG. (Only one of these paths was traversed in the original implementation.)

This path has regions in which the order that some design decisions are applied does not affect the final implementation. Consequently, we can make design decision reversals in any order. For example, we can choose the data structure before the sorting algorithm for a table, or vice versa. Thus, the order we ascend through specifications in the DAG to the LCA usually differs from the order we would descend through them to the original implementation. We exploit this commutativity to undo only the undesirable design decisions.



For example:
n_2 contains Sort-Symbol-Table
r_2 is Use-Heapsort-Algorithm
n_3 contains Sort-Symbol-Table instantiated with Heapsort-Algorithm
r'_2 is Use-Quicksort-Algorithm
n'_5 contains Sort-Symbol-Table instantiated with Quicksort-Algorithm

Figure 2. Refinement r_1 is preserved during maintenance.

A new path must then be chosen from the LCA to the desired implementation. As illustrated in Figure 2, the original design decisions r_2 and r_3 are reversed. This reversal reflects ascension to the LCA node (n_2) covering the original implementation P and the desired implementation P'. P' is then achieved through the alternative design decisions r'_2 and r'_3. This path is the descension to the desired implementation P'.

Consider our symbol table example: ascending to the LCA corresponds to reversing the decision to implement the abstraction, Sort-Symbol-Table, using the Heapsort algorithm. Descending through the r_2' refinement corresponds to choosing a different algorithm, namely, Quicksort. This method preserves all implementation design decisions made above the LCA (i.e., r_1) and thus minimizes the work required to change the program.

We can identify an LCA by (1) identifying portions of P that contribute to the undesirable behavior, and (2) reversing design decisions until the undesirable portions have been collected into a single component within a node. In Figure 3, node n_i contains the desired LCA. The component K is not at fault because, by definition, it is a semantic primitive of the domain. Thus, it is some implementation of the component that is at fault.

The refinements to be reversed are determined solely by the requirement to collect all code related to the undesired behavior into a single component. This requirement may mean reversing design decisions in parts of the program not obviously involved with the fault. In practice, this reversal is the same as making changes in one part of a program because a change made to another part requires them. Reversing refinements that do not contribute to this collection implies reimplementing parts of the program that do not need change.

As illustrated in Figure 3, the undesirable parts in the current implementation n_k

The Bettmann Archive

are collected through successive steps r_x^{-1} until a single covering component K is reached, thus defining n_i as the LCA node. In other words, the terminating condition for ascending the DAG is whether or not all parts requiring change (and their dependent fragments) have been collected in a single abstraction.

In Figure 3, component K might stand for the symbol table abstraction. Refinement r_i implements the symbol table in terms of components K_1 (which defines the data structure) and K_2 (which defines the sorting algorithm). In turn, refinement r_j implements component K_1 using some specific configuration of data structure and access methods. Presuming we need to change something concerning the implementation of the symbol table, we commence a series of collection steps beginning with the current implementation n_k. The collection step r_j^{-1} corresponds to reversing the refinement r_j. That is, all parts in the current implementation are collected within component K_1. The process continues until the termination condition is reached—that is, until all parts affected by the change are collected within a single component.

In a practical maintenance situation, a programmer uses his knowledge of the application, previous experience with similar applications, and information from the literature to identify the parts of the program responsible for the unwanted behavior. Each part, together with other pieces of coupled code, is collected into a covering abstraction. This step is repeated until all the identified parts have been abstracted into a single component, which is then reimplemented.

Achieving implementation goals

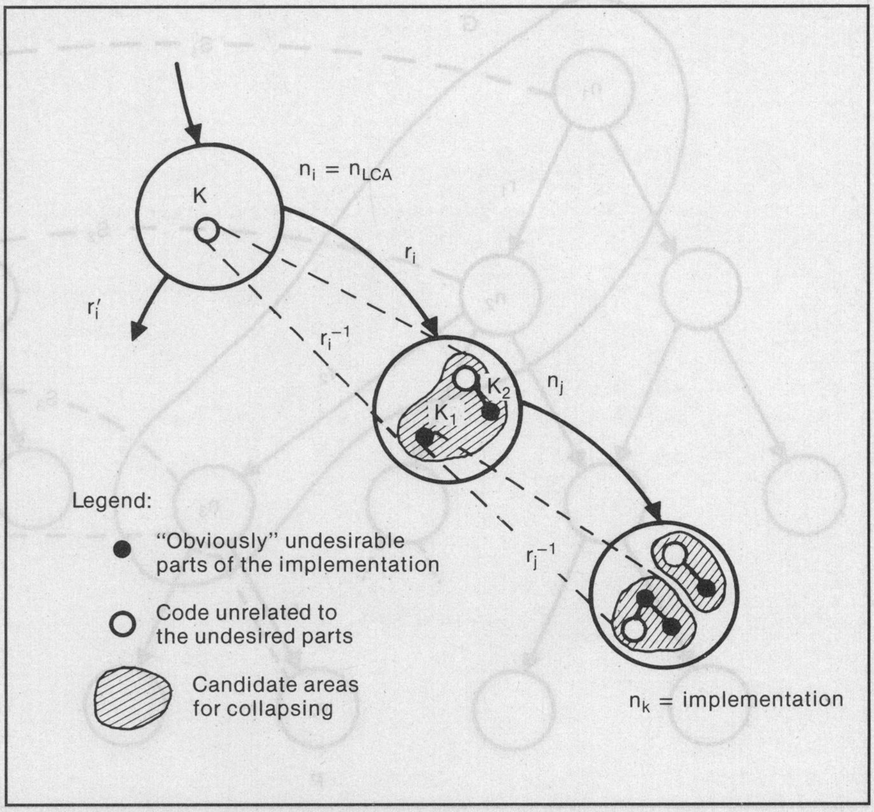
The procedures given below assume that the specification, the refinement DAG, and the implemented program are available.

Enhancing performance. Performance is changed generally by (1) changing the underlying representations used by a program and (2) using more efficient procedures made possible with the changed representation. We assume that the revised representations and corresponding procedures are already contained as refinements in the domains used to generate the current program, but they were simply not used.

If they are not already contained, then the domains must be augmented accordingly. For example, there may be reasons to change the linked list representation for the compiler's symbol table into an array structure. However, refinements for the definition and manipulation (using access or update methods or appropriate sorting algorithms) of array structure are not available. The data structures domain would then have to be augmented with the corresponding refinements.

Some nodes in the refinement DAG are LCAs that allow reimplementation of the current, low-performance abstractions. Design decisions are reversed to travel from the current implementation back to one of those LCAs. New decisions are applied to arrive at a different implementation. Within the Draco paradigm, refinement direction is changed by changing the tactics that govern the implementation decision process.

Change of environment. Changes in the environment can be accommodated in a manner similar to enhancing performance. The domains are first augmented with new refinements (r'_{new} in Figure 4) specifying how the abstractions used in those domains can be implemented by the new environment. This implicitly produces a refinement DAG that contains the original DAG plus some new possibilities introduced by the new refinements. In the compiler example, a possible requirement is that a new implementation be run on a machine with limited memory; the symbol table would therefore be in secondary storage. The data structures domain might need new refinements, for instance, to implement the Sort-Symbol-Table abstraction using external sort mechanisms. In our example, the refinement r_{new} corresponds



$n_i = n_{LCA}$
K
r_i
r'_i
r_i^{-1}
K_2
K_1
n_j
Legend:
● "Obviously" undesirable parts of the implementation
○ Code unrelated to the undesired parts
▨ Candidate areas for collapsing
r_j^{-1}
n_k = implementation

Figure 3. Finding an LCA: Collecting components into a single component.

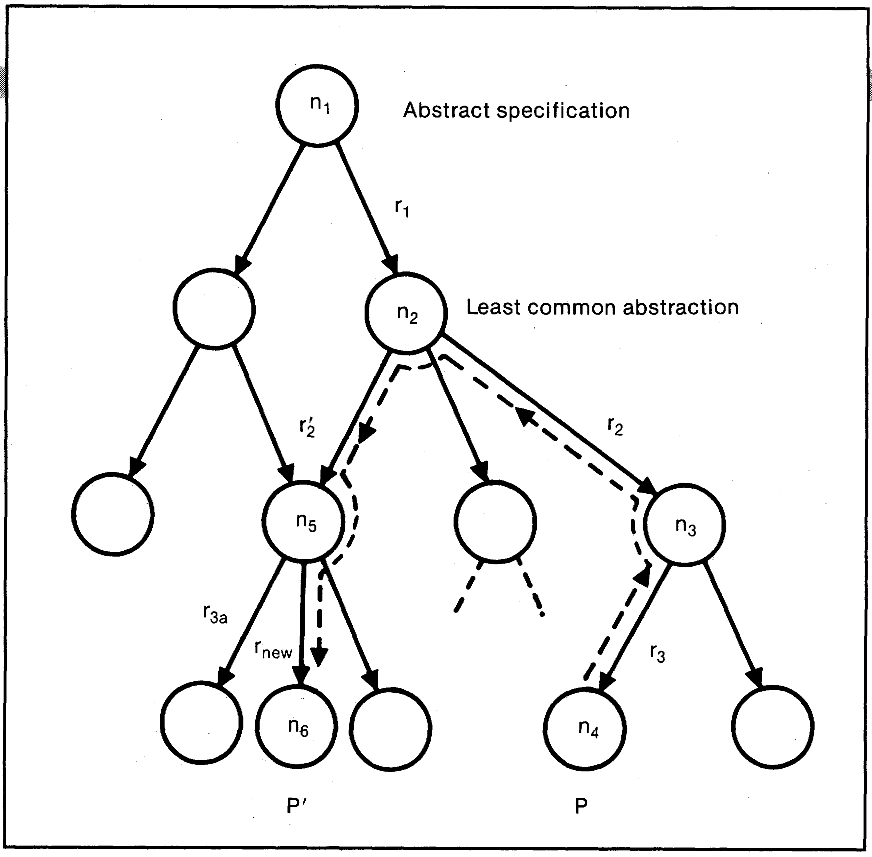


Figure 4. Changing environment: r_{new} :: = Use-Mergesort is a new refinement.

to Mergesort. A suitable LCA is found and re-refined using the new refinement.

Changing the function. The function is changed by changing the specification. Then, the new specification is simply re-refined to a particular implementation. Refining the new specification creates a new refinement DAG, as shown in Figure 5. The specification for n_1 is changed to n'_1 , and an entirely new refinement path must be followed.

A more efficient method is to determine how the DAG that implements the original specification is related to the DAG that implements the revised specification. In particular, we wish to discover a set of maintenance substitutions that relate the two DAGs. This set of substitutions must preserve the part of the design that will not change. The box at right presents the intu-

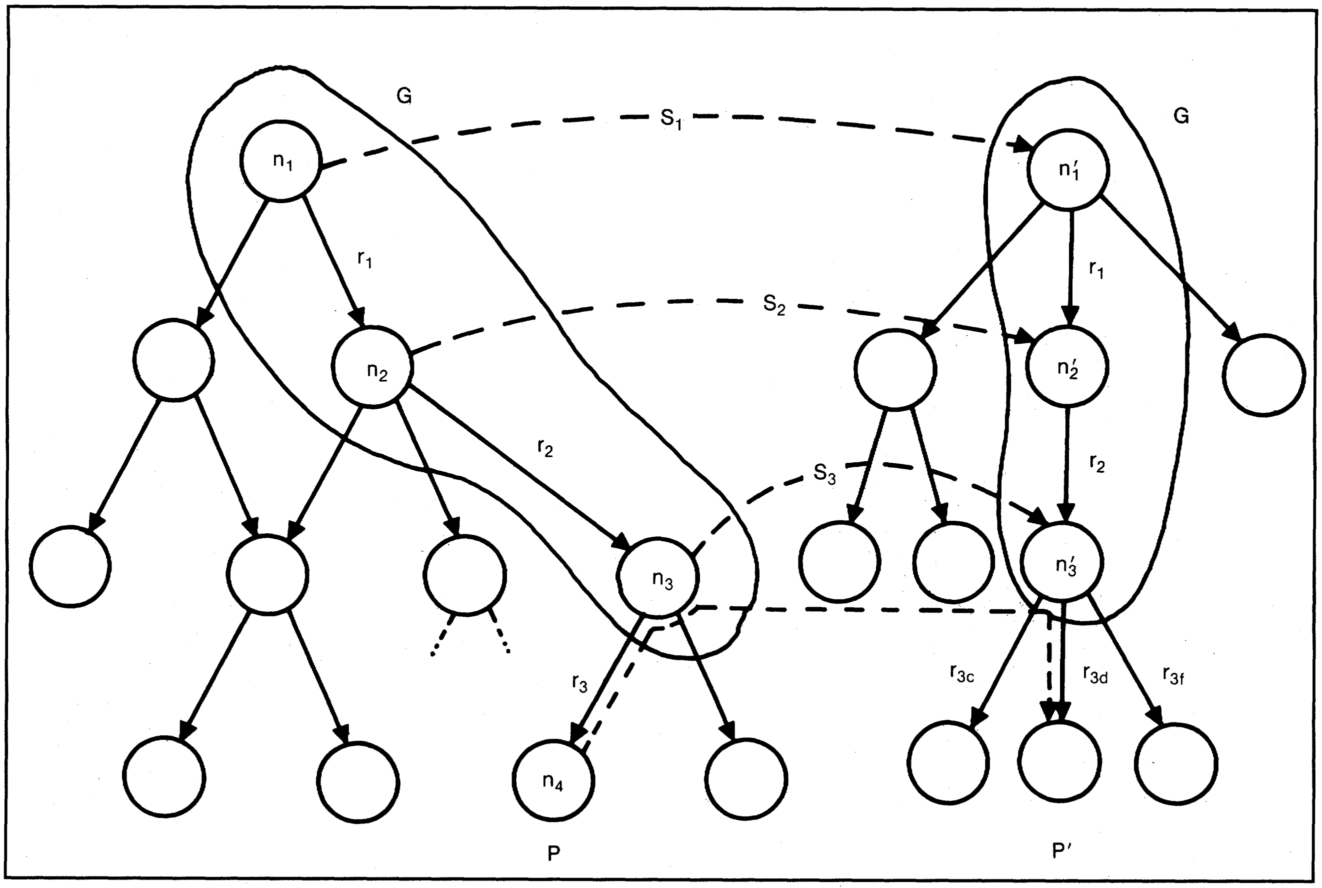


Figure 5. Changing specification: Commonality of design structure between G and G' under S, the set of substitutions S_i.

ition for the construction of a set of maintenance substitutions. A procedure for computing maintenance substitutions is available elsewhere.[1]

Given a set of maintenance substitutions S_i, we can then move up the original DAG until we find an LCA—a node where one of the substitutions may be applied. We apply the substitution, thus converting the LCA into the corresponding node in the revised DAG. Subsequent design choices are made on the path to a new implementation. (This process is indicated by the dashed line from n_4 in Figure 5.) Thus, we have avoided remaking *all* the design decisions in the common part of the two DAGs.

In a modular specification, each part of the specification has its own refinement DAG. The implementation consists of a set of leaves, one taken from each DAG. A change to the specification, then, affects only some of the specification modules, and so only some of the refinement DAGs. Leaf nodes from DAGs that do not change can be used unchanged in the new implementation. The procedure outlined in this section on achieving goals can be used to generate new leaves for the changed DAGs. Thus, modularity is seen simply as a method for making trivial the determination of the similarities on portions (the unchanged DAGs) of (what would otherwise be) a single, large refinement DAG. Hence, we can understand why well-modularized systems are easier to maintain.

Correcting a design error. A design error can be corrected by using the techniques just discussed. Design errors are either (1) failure to generate the correct specification or (2) failure to follow a correct specification. Incorrect specifications may be treated by changing functionality. Incorrect implementation corresponds to having a faulty refinement, which is easily resolved by (1) adding a corrected refinement, (2) applying the technique discussed earlier for changing the environment, and (3) removing the faulty refinement from the set of possible implementations.

In summary then, the fundamental concept in our model of maintenance is to capture the change in an LCA and reimple-

Determining maintenance substitutions

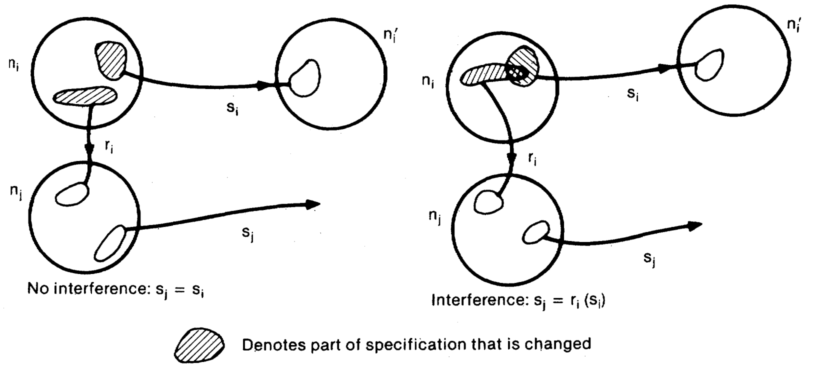
One can always determine a substitution S_1 that converts the original specification to the revised specification. (This can be constructed automatically as the original specification is revised by analyzing the editor commands.) S_1 captures the essential difference between the specifications. Both the original and the revised specification can be composed of configurations of components from several domains.

Now we determine the parts of the DAG for the revised specification that match the DAG derived from the original specification. Consider Figure 5. For each node n_i in the original refinement DAG, one can attempt to construct an S_i that converts n_i to n_i'; however, the formal methods for performing the construction are rather messy and are available in our technical report.[1] Intuitively, each S_j is related directly to the S_i of its parent and is a function of the refinement used to get from n_i to n_j. Each S_j has one of the following three properties: (1) it is not computable, (2) it is trivially computable by virtue of being identical to S_i of its parent, or (3) it is some function of S_i and r_i. Some of the S_j values simply do not exist because there is no method of using r_i on n_i'. Some of the S_j values are trivially computable because r_i does not affect the portion of the specification being changed (see figure below), and thus S_i is still applicable at n_j. If the portion of the specification undergoing change in n_i is affected by the implementation decision r_i, then S_j is clearly a function of S_i and r_i. It is easy to determine whether each S_j is computable. Furthermore, if one is computable, the actual computation is easy. The set of S_j values that are computable determine parts of the implementation DAGs that have the same design histories, G and G' (see Figure 5). Given a concrete program P, one can reverse design decisions until some node n_{LCA} in G is found (this is an LCA by definition), apply S_{LCA} to find the corresponding node in G', and apply a new design decision to implement n_{LCA}'.

Some could argue that to compute S_j one must not only have but also apply all the r_i values on the path from the root to S_j sequentially to S_1, and thus there is hardly any savings compared with applying all the r_i values to the root node itself. This is not true for two reasons.

First, presumably the S_j values are much smaller than the corresponding n_j values, resulting in the manipulation of much smaller structures than n_j. If S_j were not much smaller than the corresponding n_j, it would be more economical to reimplement the specification from scratch.

Second, many of the r_i have no effect whatsoever on S_i when applied (corresponding in practice to the fact that a change leaves parts of the software untouched). Each S_i value refers to some specific part of n_i; each r_i value refers to some other, perhaps overlapping, part of n_i (see figure below). If there is no overlap, then S_i is identical to S_j. If there *is* overlap, then S_j will be different from S_i. In other words, many refinements implement some portion of the program not affected by the change of specification. Such refinements cannot affect the various changes of specification and therefore require no energy whatsoever to apply to the various S_j values. In fact, we need not even know what they are; we need to know only that they have no effect on the change of specification. Maintenance programmers scan the entire program when told to make a change and unconsciously reject parts of it as unrelated. This corresponds to deciding that r_i does not affect the change of specification. The only refinements we must know are those that affect the change of specification, so maintenance programmers should look very carefully at the parts of code they expect to change.



n_i
n_i'
s_i
r_i
n_j
s_j
No interference: $s_j = s_i$
n_i
n_i'
s_i
r_i
n_j
s_j
Interference: $s_j = r_i(s_i)$
Denotes part of specification that is changed

The effect on S_j of r_i overlapping with S_i.

ment only the LCA. In performance and environmental change, the LCA is discovered through a sequence of collection steps. In functional change, the LCA is found by defining an appropriate set of maintenance transformations.

Recovering abstractions

The TMM rests on the supposition that the would-be maintainer has both the specification and its refinement history. But what happens if he has only the program code? Clearly, this is the scenario that maintainers face in practice. The TMM will still work in this situation, but a systematic approach must be used to recapture implementation knowledge before the TMM can be applied.

The abstraction recovery paradigm imitates what maintenance programmers probably do informally. Before making changes in a program to adapt it to new requirements, a programmer informally derives a plausible, higher level "ancestor" specification equivalent to the original program as an aid to understanding.

Figure 6 shows the conventional approach to maintenance. Arcs are represented by broken lines to indicate that the refinement history, and thus the original abstract specification, is not available. What, then, is to guide the maintainer when going from program *P* to *P*′? We propose that one should recover the design by a process known as abstraction recovery and then apply TMM. We call this approach MBA, or maintenance by abstraction.

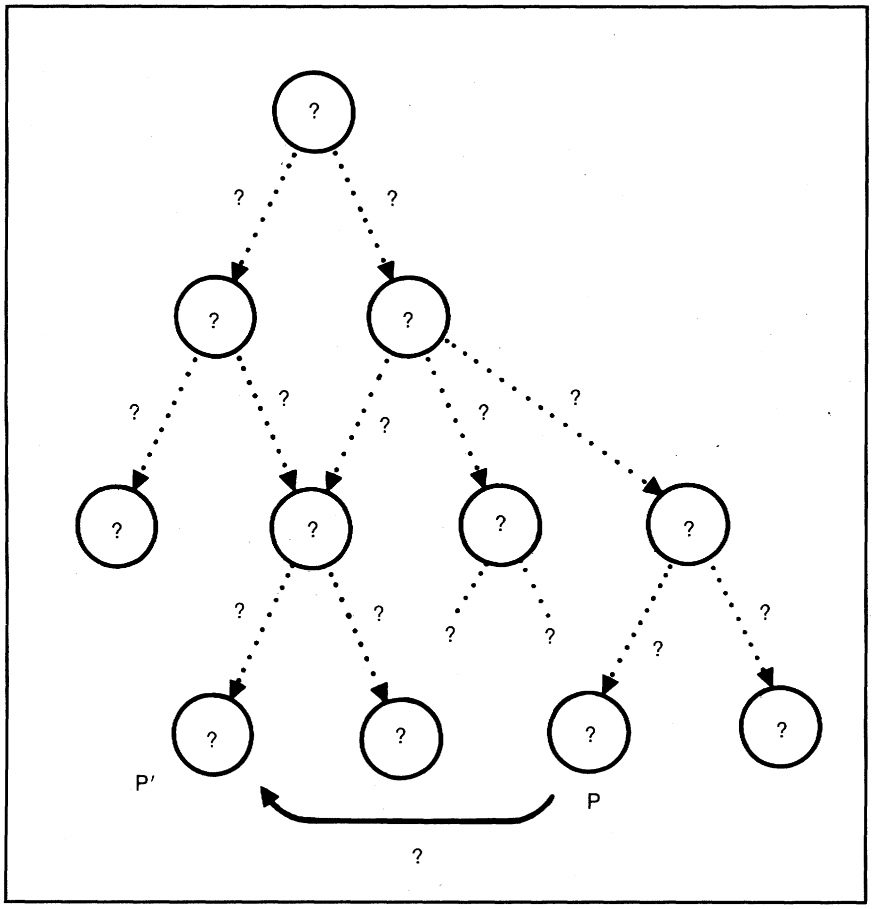


Figure 6. With little or no background in conventional maintenance, what is to guide the change process? The solution is maintenance by abstraction: First do abstraction recovery, then apply TMM.

Ancestral specifications can be developed by repeatedly performing an abstraction recovery step. Each step consists of

- inspecting the specification of interest (initially the code),
- proposing a set of possible abstractions for the program specification portion of interest,
- choosing the most suitable abstraction, and
- constructing a specification containing the new abstraction.

Each abstraction proposed implicitly selects some domains and refinements that must produce the existing code when applied to the ancestor containing the proposed abstraction. Abstraction recovery steps are repeated until a useful LCA is reached. For example, when modifying a symbol table routine, the programmer hunts for code related to symbol insertion and mentally lumps it into the abstraction Insert-Symbol. Thus a more abstract version of the actual code is formed.

Recovering a specification for program *P* causes one to navigate the refinement DAG as shown in Figure 7. First, program *P*'s plausible immediate ancestors (broken circles) are postulated. Selection of an appropriate ancestor (solid circle) is based on the conjecture that the ancestor is on the path from *P* to a suitable LCA. The path to the ancestor is taken, and the process is repeated until an LCA is reached.

The abstractions chosen should be appropriate to the purpose of the code. Good choices use domains and refinements recovered in earlier steps, perhaps with minor augmentation. In a symbol table routine, for example, one does not propose "matrix multiply" as a plausible abstraction. The maintainer's improvement through experience can be used to enhance the resulting domains.

The selection of an appropriate ancestor results from generalization based on the specification being considered. The implementation provides a very limited sample on which to base a generalization step. Recognizing this fact, Boyle suggests that refinements (their term is transformations) that codify implementation decisions are frequently irreversible.[6] Furthermore, it is impossible to determine the refinements from the program alone. In other words, "unrefinements" are possible only with additional knowledge: We must rely on the maintainer's experience, his knowledge of the application domain, input from the original designer, existing documentation, environmental specifications, etc. In practice, the junior maintainer must often ask seniors for additional information about all of these areas.

Human experience appears vital. The traditional re-creative tasks of recovering design information include drawing control-flow, dataflow, and structure charts; deriving module I/O specifications; identifying key algorithms; and interrogation. These heuristics can be used to generate possible abstractions. However, MBA does not presume any particular heuristic. When recovering a design, any method can be used.

Sneed suggests that "automation is the only true solution to the maintenance problem."[13] He argues that tools—static analyzers for module, program, and system levels of abstraction—can yield a tenfold increase in the capacity of a human being to understand and document. Sneed intimates that the output from these tools provides the raw material for the real work—generation of the system specification via abstraction.

Quite often the maintainers are not the original authors, and much time may have passed since the original implementation. Thus maintainers are likely to regenerate only approximations of the abstractions originally used. This mismatch between the maintenance DAG obtained by abstraction recovery and an actual DAG (Figure 8) is the crux of the maintenance problem. Each successive maintenance effort introduces cumulative approximation errors, making the software product more difficult to understand and modify.

Avoiding approximations and the amplification of errors through repeated maintenance is difficult. We believe that errors increase in magnitude when the recovery process is informal. The errors generated by the limited sample used for the abstraction step can be substantially

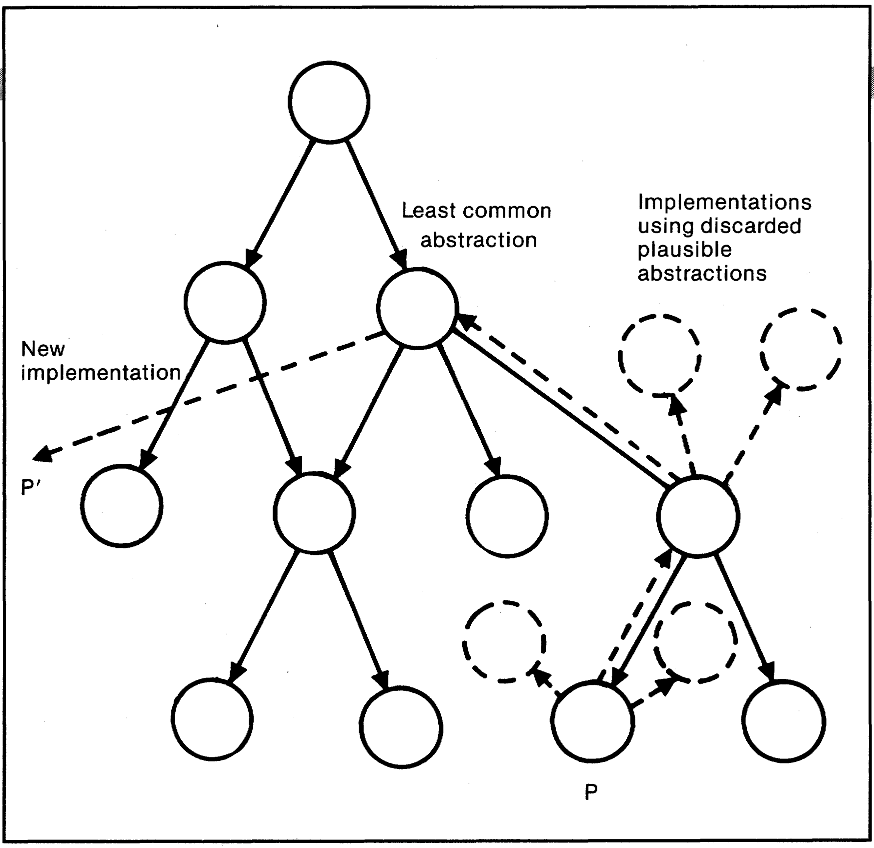


Figure 7. Abstraction recovery.

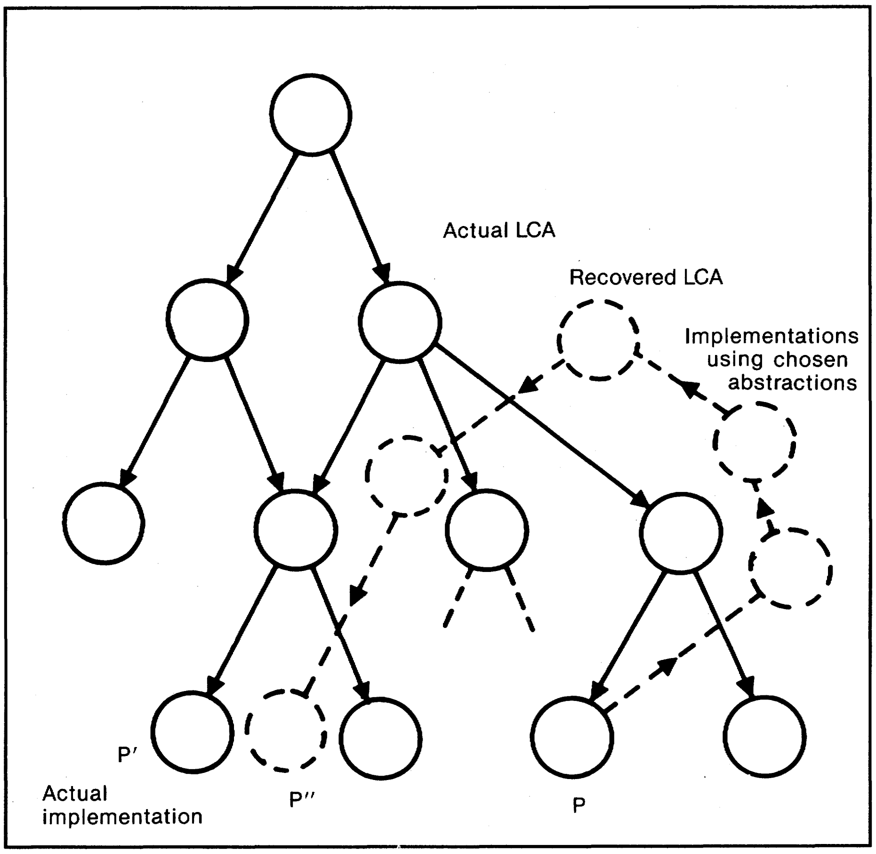


Figure 8. Recovered design versus actual design: the approximation error.

reduced by performing domain analysis prior to design recovery. Domain analysis can result in a more adequate, complete, and reusable set of domain abstractions, thus enhancing the power of the abstraction recovery paradigm.

Normally, approximation errors are undesirable. However, when a program has a poor design, approximation errors can be advantageous. Instead of recovering the specification of the poor design, one can commit an intentional approximation error that presumably leads toward a specification for a better design—a common occurrence in practical maintenance work.

It is in some sense surprising that maintenance of code is possible at all. A real maintenance activity seldom requires a complete reimplementation of a program; normally, only part of the code is changed. The refinement DAG shows that any program can have many possible paths to a particular implementation, indicating that many design decisions are commutative within regions of the implementation path—that is, the order in which refinements are applied does not affect the implementation. Refinements (design decisions) not affected can be "floated" upward within the regions of commutable refinements and need not be addressed directly in the maintenance effort. Thus, TMM, together with maintenance by abstraction, helps explain why, in practice, maintenance can be performed just on code with only knowledge of a few low-level design decisions.

Advantages and limitations

Economy of scale and product reusability are important considerations in software engineering. Some could argue that a systematic application of the maintenance-by-abstraction approach to small programs is like killing gnats with a sledgehammer. Recovering the design of a large application by our method requires reading and processing the source for the application program. Since these programs are written in conventional computer languages, capture of this information may require an effort comparable to that of writing the semantic analyzer of a compiler. While this would seem to limit the utility of the process to very big programs, such programs do exist, and we believe the potential payoff is large.

In some organizations, the recovery of the abstractions for one program may lead to the discovery of many domains that can be used to recover the designs of other programs. This amortizes the recovery costs and makes the paradigm potentially economical even for small programs.

The reuse of analysis and design information supports the economy of scale.[14] In the framework of the Draco technology, the analysis and design knowledge is formalized through networks of domain-specific languages. These languages enable software developers and maintainers to reuse the expensive analysis and design processes and to avoid a costly learning experience. Once the recovery of analysis and design information has been performed on an application, new modifications and code portings are easier, and the resources such actions require are more predictable. Since the system is formal, we can explicitly predict the effects of certain kinds of changes. Consider, for example, attempting to reimplement a system using linked lists structures instead of arrays. Currently, we cannot even predict whether such reimplementation is possible. The reuse of the abstractions recovered using our paradigm would enable us to redefine the implementation of data accesses to use lists, and then re-derive the implementation.

Boyle[6] suggests that concrete programs have a plethora of irrelevant properties that make them difficult to modify, extend, adapt, and transport. Moreover, ". . . abstract programs contain only such information as is necessary to show that they solve the problem for which they were written. Therefore, modifying, extending, adapting and transporting is much easier than it is for concrete programs."

We believe we can benefit from the economic and intellectual advantages in the reuse of the analysis and design processes, even if we have to start by recovering them from a concrete implementation. Thus, after design recovery, maintenance should be easier.

The model we have proposed is attractive because it provides a unified solution to a variety of problems associated with the need to implement changes in the functionality or performance of a system or its interfaces with the environment. Once a domain network[15] has been defined for an application area, it is easy to update programs in that area by adjusting or augmenting the refinement libraries, regenerating the applications using the new libraries, and redelivering the results to all users. This approach allows for an economic and practical configuration control and a distribution system for software applications.

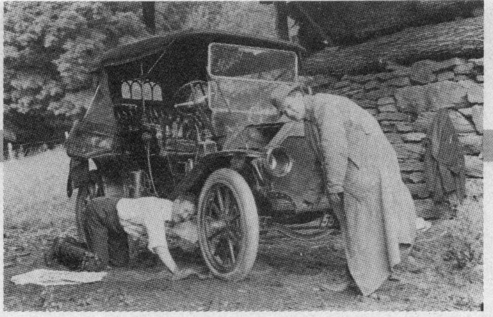
Generating the ideas

Our problem was to port a moderate-size program coded in Lisp from one machine to another. The fact that the software to be ported (Draco) happened to be the very tool we used to accomplish the port is merely a coincidence.

A principal tool in our research on the reuse of software engineering artifacts and knowledge is a prototype system that implements the Draco paradigm for the implementation of software from components. Draco was coded in UCI Lisp running on a DEC 2020. We needed to port this system because of a departmental decision to migrate from DECsystem-20 computers running TOPS-20 to DEC VAX computers running under Berkeley 4.2 Unix; consequently, we needed to change the execution environment of our program.

Since UCI Lisp is not available on this new configuration, we had to port either Draco or UCI Lisp. We quickly elected to port the smaller Draco system. We chose to retarget Draco from UCI Lisp to Franz Lisp because the latter was available under Unix, stable, and widespread in the research community.

Because one of our industry sponsors desired a VAX/VMS/Common Lisp version of Draco, we considered Common Lisp as a potential target in addition to Franz Lisp. Using Franz Lisp seemed a reasonable stepping-stone on the way to



The Bettmann Archive

creating a Common Lisp version of Draco. The prospect of producing more than one new implementation of Draco made the idea of manual conversion particularly repugnant.

The version of Draco that we moved has a complex kernel coded in uncommented Lisp, along with some specialized domains not coded in Lisp. Since the specialized domains were stated as high-level specifications, they were portable using the Draco paradigm. The kernel was the real problem.

We decided to apply the Draco paradigm to accomplish the port. (This was before the ideas on abstraction recovery had become clearer.) To minimize the impact of new code on the porting process, we imposed the following iron-clad rule: There would be no changes to the UCI Lisp source for Draco. Invocation of this rule forced us to treat the kernel as a specification.

Our first discovery (obvious in retrospect) was that Draco used only a subset of the UCI Lisp dialect. This enabled us to design a limited domain specific to the Draco functionality. Thus, we were able to effectively capture the meaning of Draco-specific UCI Lisp idioms in an abstract form and discard the concrete syntax. The recovery of the meaning from the concrete syntax is an example of reversing the design decisions to implement those abstractions with the particular UCI Lisp coding. The captured abstraction corresponds to the LCA described earlier.

The Lisp idioms captured fell into three classes:

- generic Lisp functions and S-expressions,
- UCI Lisp idioms (generally related to environmental interface, such as I/O), and
- Draco-specific abstractions implemented as procedures (Initialize, etc.).

To reimplement the Draco kernel in Franz Lisp, we coded new refinements for the abstractions captured in the previous step. (This corresponds to moving down from the LCA to a new implementation in Figure 4.) A typical example is shown in Figure 9.

The abstractions for the I/O used by Draco in UCI Lisp turned out to be difficult to refine directly to Franz Lisp. We found the semantic gap between I/O concepts in the two dialects to be too large. We were forced to define a higher level bridging domain to implement these abstractions by a virtual machine[16] technique. Some inefficiencies were introduced in the final implementation because we did not capture these components at a sufficiently abstract level. One should capture the abstractions at the highest level possible to make reimplementation easier. We expect the problem with I/O to reappear when we retarget for Common Lisp.

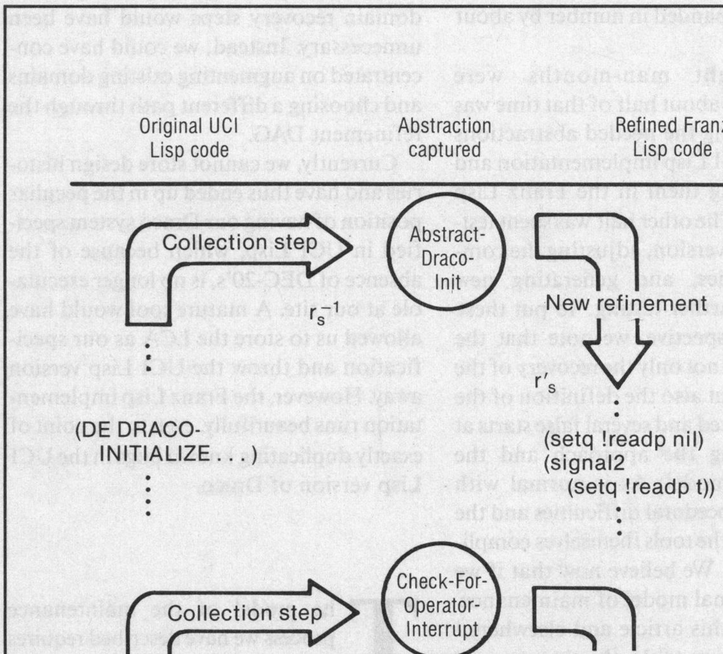


Figure 9. Example of an abstraction.

Some quantitative results

The original kernel consists of approximately 2400 lines of UCI Lisp code divided among some 170 functions. Approximately 280 abstractions were identified in four domains; refinements were implemented for each. About 45 percent of the abstractions were refined directly (most of these were generic Lisp); 14 percent of the abstractions were implemented by simulation in the target environment. The balance of the abstractions were not complex. The Draco kernel was scanned, and the UCI Lisp idioms it contained were converted into the abstractions by a UCI Lisp domain parser built especially for this purpose. This constituted the abstraction recovery step(s). Since the parser converted each part of the original specification to a different form, we effectively reversed a very large number of implementation decisions. The result of parsing was the LCA we desired. Once the original LCA was captured, we used Draco to reimplement the abstractions in Franz Lisp. The entire 2400 lines of the Draco kernel were automatically converted by the process, using 19 hours of DEC 20 CPU time. Lines of source code expanded in number by about 10 percent.

About eight man-months were expended, and about half of that time was spent recovering the needed abstractions used in the UCI Lisp implementation and reimplementing them in the Franz Lisp environment. The other half was spent testing the ported version, adjusting the component libraries, and generating new versions for further testing. To put these figures in perspective, we note that the effort included not only the recovery of the abstractions but also the definition of the domains involved and several false starts at conceptualizing the approach and the maintenance model. As is normal with prototypes, procedural difficulties and the immaturity of the tools themselves complicated the task. We believe now that if we had had a formal model of maintenance, as defined in this article and elsewhere,[1] and more mature tools, the time to port would have dropped by a factor of approximately four.

The porting itself went easily in spite of our lack of familiarity with the actual design of the ported code. Even now we do not have a deep understanding of the ported code; complex meta-pattern matchers in the Draco kernel were automatically converted without our looking at that code at all. Most of the bugs discovered while testing the Franz Lisp version were easy to find, relatively easy to fix, and had to be discovered/fixed only once, as the fixes were automatically propagated by the refinement process.

We could probably have moved a much larger program without much additional effort; only new semantic primitives would have required our attention. Porting different UCI Lisp programs would similarly require intervention only for new semantic primitives. We expect that much of the human effort already expended can be reapplied if we decide to proceed with a Common Lisp implementation. Additionally, should some bug be discovered in the Draco kernel, we need only modify the kernel and regenerate both a Franz Lisp and a Common Lisp system automatically. Had the original program been constructed using the Draco paradigm, the domain recovery steps would have been unnecessary. Instead, we could have concentrated on augmenting existing domains and choosing a different path through the refinement DAG.

Currently, we cannot store design histories and have thus ended up in the peculiar position of having our Draco system specified in UCI Lisp, which because of the absence of DEC-20's, is no longer executable at our site. A mature tool would have allowed us to store the LCA as our specification and throw the UCI Lisp version away. However, the Franz Lisp implementation runs beautifully, even to the point of exactly duplicating known bugs in the UCI Lisp version of Draco.

The model of the maintenance process we have described requires (1) program specifications at some abstract level and (2) the set of design decisions that were made to implement the program. The TMM can be formalized, and effective tools to aid software maintenance can then be constructed for programs built of components. The scheme called "maintenance by abstraction" is intended to help those maintainers unlucky enough to possess only concrete code. MBA suggests that a human with appropriate tools can perform abstraction recovery to (partially) convert such programs into their constituent components, and then changes can be made to the resulting design using the TMM. We have shown the utility of this approach by successfully porting a real, complex software system.

Now that we have a theory supported by positive application experience, tools could be developed to aid this approach and significantly enhance the practice of software maintenance. □

Acknowledgments

We gratefully acknowledge the support of the National Science Foundation (MCS-83-04439) as well as joint sponsorship of the University of California and the Aluminum Company of America under the Micro program.

References

1. G. Arango et al., *A Formal Model of Transformation-Based Software Maintenance*, Advanced Software Engineering Project Technical Report RTM-39/86, University of California, Irvine, 1986.
2. J. M. Neighbors, *Software Constructions Using Components*, Technical Report TR-160, University of California, Irvine, 1980.
3. J. M. Neighbors, "The Draco Approach to Constructing Software from Reusable Components," *IEEE Trans. Software Eng.*, Vol. SE-10, No. 5, Sept. 1984, pp. 564-573.
4. J. M. Neighbors, J. Leite, and G. Arango, *Draco 1.3 Users Manual*, RTP003.3, University of California, Irvine, June 1984.
5. S. Hague and B. Ford, "Portability—Prediction and Correction," *Software: Practice and Experience*, Vol. 6, No. 1, John Wiley & Sons, New York, 1976, pp. 61-69.
6. J. M. Boyle and M. N. Muralidharan, "Program Reusability Through Program Transformation," *IEEE Trans. Software Eng.*, Vol. SE-10, No. 5, Sept. 1984, pp. 574-588.
7. M. Shaw, "Abstraction Techniques in Modern Programming Languages," *IEEE Software*, Vol. 1, No. 4, Oct. 1984, pp. 10-26.
8. A.S. Tanenbaum, P. Kling, and W. Bohm, "Guidelines for Software Portability," *Software: Practice and Experience*, Vol. 8, No. 6, John Wiley & Sons, New York, 1978, pp. 681-698.

9. P.C. Poole and W. M. Waite, "Portability and Adaptability," in *Advanced Course on Software Engineering,* F.L. Bauer, ed., Springer-Verlag, Berlin, 1973.

10. B. W. Boehm, *Software Engineering Economics,* Prentice-Hall, Englewood Cliffs, N.J., 1981, pp. 54-55.

11. A. Partsch and R. Steinbruggen, "Program Transformation Systems," *Computing Surveys,* Vol. 15, No. 3, Sept. 1983, pp. 199-236.

12. N. Wirth, "Program Development by Stepwise Refinement," *Comm. ACM,* Vol. 14, No. 4, Apr. 1971, pp. 221-227.

13. H. M. Sneed, "Software Renewal: A Case Study," *IEEE Software,* Vol. 1, No. 3, July 1984, pp. 56-63.

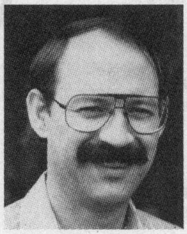
14. P. Freeman, "Reusable Software Engineering: Concepts and Research Directions," *Proc. ITT Workshop Reusability in Programming,* Stanford, Conn., 1983, pp. 2-16.

15. G. Arango and P. Freeman, "Modeling Knowledge for Software Development," *Proc. Third Int'l Workshop Software Specification and Design,* IEEE-CS Press, Los Alamitos, Calif., 1985, pp. 63-66.

16. R. A. Meyer and L. H. Seawright, "A Virtual Machine Time-Sharing System," *IBM Systems J.,* Vol. 9, No. 3, 1970, pp. 199-218.

Ira D. Baxter is working on a PhD in computer science at the University of California, Irvine, where he received BS and MS degrees in 1973 and 1983, respectively. A member of the Advanced Software Engineering Project, his research interests include machine learning, portable software, operating systems, compilers, machine architectures, and the application of AI principles to software and hardware engineering. He has worked extensively in industry, designing/implementing two complete multiuser operating systems plus tools, as well as a 16-bit virtual memory computer.

Baxter currently consults for industry through his company, Software Dynamics, in Anaheim, California. He is a member of IEEE, IEEE-CS, ACM, and the American Association for Artificial Intelligence.

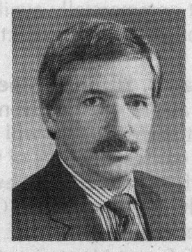


Christopher W. Pidgeon is a PhD student in the information and computer science program at the University of California, Irvine. He has been an associate professor of computer information systems at California State Polytechnic University since 1979. Within the Advanced Software Engineering Project, his current research focuses on the decision processes in software design. He has coauthored a textbook, *Structured Analysis Methods.*

Pidgeon earned a BS in business administration and an MBA from California State Polytechnic University, Pomona, and holds an MS in computer science from the University of California, Irvine. He is a member of ACM and the IEEE Computer Society.

Guillermo Arango is a PhD student in computer science at the University of California, Irvine. Previously, he was on the faculty of the University of Belgrano, Argentina. He has also been a systems developer and a consultant in real-time software. As a member of the Advanced Software Engineering Project, his focus is on formalizing the analysis and representation of domain knowledge to support software systems development and evolution.

Arango holds an MS in computer science and a BS in mathematics. He is a member of the IEEE Computer Society, ACM, and the American Association for Artificial Intelligence.



Peter Freeman has been a faculty member at the University of California, Irvine, since 1971. From 1969 to 1971, he did postdoctoral work at Carnegie-Mellon University. He has also worked with the United Nations in Hungary and the Philippines. He currently leads a group developing advanced concepts and tools for application in the early stages of complex system development. In addition, he serves on the Alcoa Science and Technology Advisory Council, is a consulting editor for McGraw-Hill, and is a member of the academic review board for the IBM Corporate Technical Institutes.

Freeman graduated from Rice University in 1963 with a bachelors degree in physics, from the University of Texas at Austin in 1965 with a masters degree in mathematics, and from Carnegie-Mellon University in 1970 with a PhD in computer science.

The authors can be contacted at the Dept. of Information and Computer Science, University of California, Irvine, Irvine, CA 92717.