Creating Abstract Superclasses by Refactoring

William F. Opdyke
AT&T Bell Laboratories
Naperville, Illinois 60566
opdyke@iexist.att.com

Ralph E. Johnson
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801
johnson@cs.uiuc.edu

Abstract: This paper focuses on object-oriented programming and one kind of structure-improving transformation (refactoring) that is unique to object-oriented programming: finding abstract superclasses. We decompose the operation of finding an abstract superclass into a set of refactoring steps, and provide examples. We discuss techniques that can automate or automatically support these steps. We also consider some of the conditions that must be satisfied to perform a refactoring safely; sometimes to satisfy these conditions other refactorings must first be applied.

1 Introduction

Software systems tend to grow with age as new features are added and old features are changed. It is not just the increased number of features that makes the system grow; most of us have rewritten an old system and seen its size shrink dramatically. The real problem is that the original design was not suited for the subsequent purposes of the system. Also, programs are often extended defensively, by copying code instead of changing the original version. For example, instead of making a subroutine more general, a programmer might make a new version of it and call that version, and so avoid any chance of affecting other parts of the system that called the original subroutine. The result is that a program's structure deteriorates over time.

The only way to prevent the structure of programs being maintained from decaying is to rewrite them. This paper is about techniques for incrementally rewriting programs and improving their structure. It focuses on object-oriented programming and one kind of structure-improving transformation that is unique to object-oriented programming: finding abstract superclasses.

Object-oriented programming is touted as being more reusable and extensible than conventional programming [19]. Nonetheless, the structure of an object-oriented program still deteriorates as features are added. As an object-oriented program grows, class hierarchies get larger and less rational, code is duplicated, and individual classes get larger and harder to understand. A common practice in the object-oriented community is to interleave periods of growth with consolidation periods in which the program is *refactored* (restructured) to make it smaller and easier to understand [18].

Refactoring is also important in developing reusable software, especially frameworks [14, 20, 21]. Frameworks are program skeletons that can be "fleshed out" to construct a complete program [9, 30]. Frameworks are usually developed by generalizing a set of concrete applications. Sometimes frameworks are developed by careful planning, starting with a domain analysis and a study of several applications in the problem domain. In these cases refactoring is less important. However, if existing software is going to reflect the new abstraction then it has to be refactored. Also, it often is more economical to refactor an existing application to extract a framework that could have been used to create it than to start over. Finally, an important part of building a framework is testing it for reusability by building applications that use it, and these test cases often point out the need for changing the framework. In these cases, understanding refactoring is crucial.

Refactoring has always been carried out manually, but there have been several studies of how to automate or partially automate it. Casais [7] and Berg-

stein [5] have both invented algorithms to create abstract classes from a set of concrete classes. These algorithms do the easy part of abstraction, which is moving common features to a single class. The hard part of the process is deciding whether two features are the same, and making similar features be the same. Casais has done some work in this latter area; we consider several issues outside the scope of his work.

This paper looks at one particular refactoring that is unique to object-oriented programming: making an abstract superclass of a set of concrete classes. It should be noted that any kind of program can be refactored, not just object-oriented ones, though the set of possible refactorings depends on the style of the program. For example, Griswald refactored Scheme programs [12]. In fact, though he didn't study the object-oriented transformations that are the subject of this paper, he showed how to implement some of the simpler refactorings that we will use, such as converting code segments into procedures.

Refactoring to create an abstract superclass is more complicated than has been recognized in the past, and depends as much on the purpose of the abstraction as in the structure of the program. Thus, it is unreasonable to try to completely automate it. Our interest in this problem is to provide tools for making refactoring easier, so we want to automate as many steps in the refactoring as possible. Below we describe what can be automated, what cannot be automated, and what that are still up in the air.

2 Examples of Finding Abstract Classes

An abstract class is a class designed to be used *only* as a superclass [30]. This is in contrast to the normal way that a class is used, which is both by making instances of it and by using it as a superclass. An abstract class usually defers the implementation of some of its operations to its subclasses, so it is only a partial specification of an object and cannot be directly used to make instances.

Abstract classes are an important design technique, but are not always directly supported by object-oriented programming languages. Although statically typed languages such as C++ often have a way to give a signature for an operation without giving an implementation (e.g. pure virtual functions in C++ [10]), untyped languages such as Smalltalk specify abstract classes only by convention. In Smalltalk, for example, documentation usually specifies which classes are abstract, the names of abstract classes sometimes start with "Abstract", and the operations in an abstract class that are left to subclasses are supposed to be implemented to generate a "subclass responsibility" error message.

Abstract classes are always invented by generalizing from concrete subclasses. Once an abstract class is found, many more concrete subclasses can be made from it in a top down fashion, but the original discovery of an abstract class is bottom up. This bottom up discovery can happen early in the life-cycle of a system: Wirfs-Brock et. al. show how to find abstract classes during an early design phase before any algorithms have been specified [29]. However, abstract classes can be discovered anywhere in the life-cycle of a system, though pulling abstract classes out of a set of concrete classes is harder once the classes have been implemented, as the next two examples (using C++ syntax) show.

2.1 Matrix Example

This first example shows what happens when a Matrix class is generalized to support sparse arrays. We will start with a concrete Matrix class is that is not sparse, then build a sparse version, and then capture their commonalities in an abstract Matrix class. Consider this initial implementation of the class Matrix:[1]

```
class Matrix
   {
   protected:
      int elements[10000];
      int columns, rows;
   public:
      int get(int rowNum, int colNum)
        { ... };
      void put (int newVal, int rowNum,
        int colNum) { ... };
      Matrix (int numRows, int numCols)
        { ... };
      Matrix matrixMultiply (Matrix m2)
        { ... };
      void rotate() { ...};
      Matrix martixInverse() { ...}
   };
```

A reference to, for example, the matrix element in row x, column y might be coded as:

```
j = elements[(x * columns) + y]
```

The current implementation is fine for dense matrices, but a different representation would be more

[1] In this example, the elements of a matrix are always integers.

space efficient for sparse matrices. Such a representation would store only the non-zero values, along with their locations. Retrieving and storing elements would be different with this representation. The following steps could be applied to support both types of matrices, while capturing their commonalities in an abstract superclass:

1. rename the Matrix class to be denseMatrix
2. in functions other than *get* and *put*, replace references to *elements* with calls to *get* and *put*.
3. define a new class sparseMatrix, and copy the members of denseMatrix into sparseMatrix
4. define a new type (*sparseElement*), to represent an element of a sparse matrix. Each sparseElement stores its location along with its value.
5. in the class sparseMatrix:
   (a) change the type of the variable *elements* to be:

   ```
   sparseElement elements[50];
   ```

   (b) change the *get & put* functions. For a sparse matrix, the function *get* will return 0 if no element is defined in *elements* for the specified location. The *put* function will remove the old value (if any) from *elements* and write a new value there only if the value is non-zero. Since the other functions are written in terms of the *get & put* functions, they need not be changed.
6. define an abstract superclass Matrix for classes denseMatrix and sparseMatrix:
   (a) add to Matrix class the function signatures for *matrixMultiply, rotate,* and *martixInverse*. These define the *protocol* for the class; that is, the set of messages that an instance of the class will accept.
   (b) move the variables *columns* and *rows* to the superclass
   (c) add the function signatures for *get & put* to the superclass
   (d) add the function bodies for *matrixMultiply, rotate,* and *martixInverse* to the superclass, and delete the redundant definitions from the subclasses.

After these changes, class Matrix is an abstract superclass. Its only data are variables to store the number of columns and rows, but the data in the Matrix is defined by the subclasses. Matrix defines *get* and *put* as pure virtual functions, so their implementation is left to subclasses, too. However, it can define operations such as *matrixMultiply*, *rotate*, and *matrixInverse* in terms of *get* and *put*.

Classes denseMatrix and sparseMatrix are concrete subclasses of Matrix that define how the data for the matrix is stored and implement *get* and *put* functions. These classes will also have to implement constructor and destructor functions. In practice, some of the algorithms inherited from class Matrix might be too inefficient and will probably be reimplemented, but the original algorithms are correct. For example, most representations of sparse matrixes permit a more efficient way of rotating the matrix than just iterating over all the elements and relocating them, which is the natural algorithm to be defined in class Matrix.

In summary, as a result of these refactorings the program defines two types of Matrices, making explicit their common features. This structure would make it easier to extend the program to support additional matrix representations in the future.

This example assumes that the programmer recognized early that the storage and retrieval operations would differ for dense and sparse matrices, and replaced direct references to *elements* with calls to the functions *get* and *put*. The next example shows a case where code replacement is done later in the refactoring process.

2.2 Inode Example

The second example describes how refactorings were applied to improve the Inode[2] class during the design of the *Choices* file system framework [17]. An Inode contains a description of the disk layout of a file and other information such as the file owner, access permissions and access times. The *Choices* object-oriented operating system project at the University of Illinois has defined an operating system framework consisting of interlocking frameworks for file systems [17], virtual memory [26], communication [31], and process scheduling [25]. An early version of the *Choices* file system framework supported only the BSD UNIX file format. Then, it was extended to handle both BSD UNIX and UNIX System V [1] file formats. To support both formats, the Inode class was changed as follows:

1. the Inode class was renamed to BSDInode,

[2] Inode is standard UNIX® operating system terminology; it is a contraction of the term *index node*. UNIX is a registered trademark of UNIX Systems Laboratories, Inc.

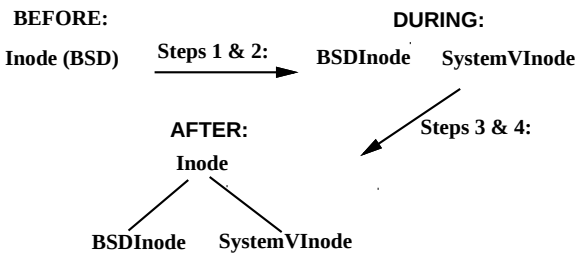

Figure 1: Creating An Abstract Superclass

2. the SystemVInode class was added as a sibling of BSDInode[3]; variables and functions were copied from the BSDInode class, and modified,
3. a new class Inode was added as the superclass of BSDInode and SystemVInode,
4. the members common to BSDInode and SystemVInode were migrated up to their common superclass.

The fourth step had to make structural modifications to the subclasses before some of the common members could be moved. For example, while most of the code in the SystemVInode implementation of the *mapUnit* function was the same as in the BSDInode class, there were a few minor differences. Getting and setting logical block numbers was handled differently for the two file formats. To handle this, the differing code was first split off into separate functions; then, when the implementations of the *mapunit* function in both subclasses matched, it was moved to the superclass.

3 Steps In Creating An Abstract Superclass

This section will show how to create a abstract superclass for a pair of classes A and B. It is easy to generalize this to more than two classes. A and B must already have a common superclass, or no superclasses. If necessary, one of them can be moved in the superclass graph so that they become sibling classes [20].

The first important step in creating an abstract superclass S of a pair of classes A and B is to create an empty class with a unique name that is a sibling of A and B. Then, A and B are given S as a superclass.

[3] *Sibling* classes are classes that either share a common direct superclass or are both top most classes in their inheritance hierarchy.

The prior examples (in particular, step 6 of the Matrix example and step 4 of the Inode example) illustrate the subsequent refactoring steps involved in creating an abstract superclass, which are defined below [20]:

- adding function signatures to the superclass protocol (after making them compatible in both subclasses)
- making functions bodies (and the variables referenced by them) compatible in both subclasses
- migrating common variables to the superclass
- migrating common code to the superclass.

3.1 Adding Function Signatures To The Superclass

Functions belong in the superclass protocol if they are part of the common abstraction represented by the superclass. Sometimes, function signatures in one or both of the classes need to be changed before the signature can be added to the superclass. Suppose, for example the abstract superclass Vehicle is created for the existing classes Automobile and Submarine. Class Automobile has a function whose signature is:

```
void shiftDirection(direction newDirection,
                    int newSpeed)
```

while in class Submarine there is a function whose signature is:

```
void redirect(int newSpeed,
              direction newDirection).
```

For the function signatures to match, one of the function names needs to be changed, and function arguments reordered. There is a range of assistance that we could expect from a tool. Hueristics could be applied to determine, based on structural attributes of the function signatures, what refactorings would be needed to make them match [20]. In the above example, the tool could prompt the user that a renaming was needed, and provide a menu of choices: one of the functions could be renamed to match the other function, or both could be given an (identical) new name. Similar support could be provided for argument reordering.

A more powerful form of automated support would be to determine, given two classes, what functions have structural similarities that suggest conceptual similarities. However, hueristics based on structural

similarities are not foolproof. Suppose that the classes Automobile and Submarine each contained a function with a single, integer argument. But, suppose that in the Automobile class, the function is called *changeOil* whose argument is the number of quarts of oil needed; in the Submarine class, the function is called *submerge* whose argument is the depth to which to descend. These functions clearly don't share a common abstraction, but automatically matching on attributes of the signature won't detect this.

Such hueristics, despite their shortcomings, may be powerful enough to support practical refactoring tasks. More powerful similarity detection is possible in some cases [8, 11].

Once the signature of a function in both subclasses match, the function signature can be added as to the superclass.[4]

3.2 Making Function Bodies Compatible

As for the *mapunit* functions described earlier in the Inode example, the function bodies in the subclasses may be similar but not identical. Before the function body can be migrated to the superclass, differences need to be separated from the common code.

The approaches for detecting program differences involve string comparison, tree comparison or a combination of these techniques [4]. The approaches represent the differences between programs as a set of edit operations (code insertion, replacement and deletion), to get from one program to the other. Program differences have been studied in regard to spelling correction [13, 27], parsing error correction [28], version storage [24] and other uses. String comparison finds the minimum cost sequence of edit operations to convert one string into another. Tree comparison algorithms detect syntactic differences between programs by building syntax trees and comparing the trees. Tree comparison algorithms are more expensive than string comparison approaches, but are not as sensitive to minor differences in coding style (for example, extra spaces or blank lines).

These techniques can be used in refactoring as follows [20]: Suppose the function *commonFunction* is defined in classes C1 and C2. For each edit operation, define a new function in both classes:

- for each insertion, define in C2 a new function whose body contains the inserted code; define in C1 a new function with the same name whose body is null. Convert in C2 the inserted code to a call to the new function; at the corresponding location within *commonFunction* in C1, add a call to the new function.

- for each replacement, define in C1 a new function whose body contains the replaced code; define in C2 a new function with the same name whose body contains the replacing code. In C1, convert the replaced code to a call to the new function; in C2, convert the replacing code with a call to the new function.

- for each deletion, in C1 define a new function whose body contains the deleted code; define in C2 a new function with the same name whose body is null. In C1, convert the deleted code to a call to the new function; at the corresponding location within *commonFunction* in C2, add a call to the new function.

These operations are safe if the code segments being inserted, replaced and deleted make syntactic sense as the bodies of new functions. This can be more easily realized by using tree analysis approaches.

While such function splitting can be safely applied to a program, the resultant new functions will not necessarily correspond to meaningful concepts in the application domain. For example, when two functions are compared, segments of code that differ between them may be preceded or followed by segments of related code that (coincidentally) are the same in both functions. In order for the new functions to represent meaningful abstractions, this "common" code might really belong together with the differing segments in those new functions.

This suggests that automated analysis should be combined with user interaction, such as the approach Rak [23] describes for abstracting a function (Smalltalk method) into a superclass from its subclass implementations.

3.3 Moving Variables

Having created the abstract superclass and determined the function signatures, it is sometimes necessary to add member variables to the abstract superclass. The most common reason is that they are referenced by common code that belongs in the superclass.

As was the case with function signatures, variables defined in one subclass may be structurally similar to, but not exactly match, conceptually equivalent variables in the other subclass. As with function signatures, structural heuristics could detect structural

[4] In C++, the function can be defined as a pure virtual function by assigning its value to be '0' in the superclass.

similarities (in name, access control mode and type). In cases where the attributes of the variables differ, refactorings can be applied to make them conform [20].

Once the attributes of a variable in both subclasses match, the variable can be moved to the abstract superclass.

3.4 Migrating Common Code to the Abstract Superclass

Before migrating the function body to the superclass, any differences between the functions need to be determined. Variables and functions referenced by the common code must be visible from the superclass before the common code can be moved there.

The preconditions for this refactoring are:

1. the function signature, but not the function body, is already defined in the superclass[5]
2. each differing code segment can be converted to a legal function
3. the scope of all variables and functions referenced by the common code includes the superclass and both subclasses.

After checking its preconditions, this refactoring:

1. for each differing code segment:
   (a) creates a new function in each subclass. The name of the new function is automatically generated and is distinct from the name of any existing member.[6]
   (b) adds the signature of the new function to the superclass protocol
2. adds a function body to the member function signature in the superclass
3. deletes the member function from the subclasses.

3.5 Summary

Given two classes, this section defines an approach for creating a common abstract superclass that contains a set of member function signatures, and possibly a set of member variables and the partial implementations of some functions. After refactoring, the definitions in the subclasses are streamlined, as some of the behavior that had been locally defined is now inherited. The commonalities and differences between the subclasses are made more explicit.

[5] This is satisfied by the results of a prior step.

[6] A refactoring could later be applied to make the name more descriptive.

4 Conclusions

Although refactoring is common in the Smalltalk community, it seems to be practiced less often by the C++ community and does not seem to be recognized as an important part of the software life-cycle of C++ programs. We conjecture that one important reason is that refactoring has been easier to do for Smalltalk programs than for C++ programs. Smalltalk is a simpler, more compact language than C++. The browsing and cross-reference tools that make refactoring easier are more a part of the Smalltalk programming environments than among some currently used C++ programming environments. However, as C++ programming environments are becoming more powerful refactoring is becoming easier to realize for C++ programs.

Refactoring seems to be considered even less important outside the object-oriented community. This is probably because its relative cost and benefit differ from one group to another. Reducing the cost of refactoring should also encourage these groups to consider refactoring their programs to keep the programs well-structured and make it easier to produce reusable software.

There are several reasons why refactoring is hard. The first is that it isn't recognized as significant. We find that simply having names for the different refactorings makes it easier to notice when they are needed, to plan for them, and to carry them out. Another reason is that refactoring takes time. It requires analyzing the program to find all the places that have to be changed. Refactorings that require many simple changes can still take a long time to carry out by hand. A third reason is that any change to a program, including a refactoring, can introduce defects into it.

We have addressed the first problem by specifying a set of refactorings that are commonly used in the object-oriented community [20, 21]. These refactorings include low-level transformations such as changing the name of a function or variable, moving a function or variable from one class to another, and breaking a function into smaller pieces, and also contains higher-level refactorings such as dividing a class into several smaller classes and finding abstract superclasses. We have prototyped most of these refactorings for a constrained set of C++ programs, and are in the process of building a refactoring tool for Smalltalk.

The best way to solve the last two problems is to provide tools to carry out refactoring automatically. Unfortunately, it is probably impossible to completely automate refactoring. The purpose of a refactoring

is to improve the design of a system, but a refactoring that can be applied safely to a program will not necessarily improve its design. On the contrary, applying arbitrary refactorings to a program is more likely to corrupt the design rather than improve it, even though the behavior of the program is unchanged. A refactoring improves design if the resultant code units correspond to meaningful abstractions that make it easier to refine or extend the program. What abstractions are meaningful depends on the application and on the designer. This implies that refactoring tasks, especially the more complex tasks, require some interaction with the designer. Nevertheless, much support can be provided by a refactoring tool [20].

We have algorithms for all of our refactorings, though many of these algorithms require several inputs from a user. Each algorithm has a precondition; if the precondition is met then the algorithm is behavior preserving and will not introduce any defects into the program. Since testing for program equivalence is undecidable, the preconditions are often conservative. Several of them are based on dataflow techniques, and can almost certainly be improved upon. However, the undecidability of the basic problem requires that any algorithm for checking preconditions will be too conservative.

Refactoring is similar to the schema modification in databases [3, 16, 22]. The main difference is that schema modification is concerned only with data, while refactorings are concerned with both data and program. On the other hand, the work on schema modification is concerned with updating the existing objects in the database. Our work has ignored the problem of changing existing objects, since most implementations of languages such as C++ do not allow programs to be modified in the middle of their execution. An OODBMS, which unifies programs and persistent data, would have to deal with both.

Refactoring is also similar to the more traditional program transformation work, which usually has the goal of improving efficiency, of converting abstract program schemas into code, or of transforming an abstract design into a concrete program [2, 6, 15]. These systems often perform the inverse transformations to ours, since refactorings often are used to make programs more abstract and are not usually concerned with efficiency.

Refactoring is a practical problem that needs better support. The examples in this paper show that even relatively straightforward refactorings such as finding a common superclass are more complicated than they appear at first. Although this seems to eliminate chances for refactoring programs completely automatically, it should be possible to build tools that make refactoring easier. In the long run, this will help make our programs easier to extend and will make it easier to develop reusable software.

5 Acknowledgements

Peter Madany provided helpful input regarding the evolution of the *Choices* file system framework. Janet Coleman, Warren Montgomery and Ed Rak reviewed drafts of this paper. The conference reviewers also provided helpful comments.

AT&T Bell Laboratories has supported William F. Opdyke's research at the University of Illinois under the full-time doctoral support program.

References

[1] AT&T. *UNIX System V User Reference Manual.* AT&T, 1984.

[2] Robert Balzer. A fifteen-year perspective on automatic programming. In *Software Reusability - Volume II: Applications and Experience*, pages 289–311, 1989.

[3] Jay Banerjee and Won Kim. Semantics and implementation of schema evolution in object-oriented databases. In *Proceedings of the ACM SIGMOD Conference*, 1987.

[4] Carol Sue Beckman-Davies. *Finding Program Differences Based on Syntactic Tree Structure.* PhD thesis, University of Illinois at Urbana-Champaign, 1989.

[5] Paul L. Bergstein. Object-preserving class transformations. In *Proceedings of OOPSLA '91*, 1991.

[6] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

[7] Eduardo Casais. *Reorganizing an Object System*, pages 161–189. Centre Universitair d'Informatique, Universite de Geneve, 1989.

[8] N. Dershowitz. Programming by analogy. *Machine Learning: An Artificial Intelligence Approach (R.S. Michalski, J. G. Carbonell and T. M. Mitchell, eds)*, 2:395–424, 1986.

[9] L. Peter Deutsch. Design reuse and frameworks in the Smalltalk-80 system. In *Software Reusability - Volume II: Applications and Experience*, pages 57–72, 1989.

[10] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Co., Reading, MA, 1990.

[11] R. Greiner. Learning by understanding analogies. *Artificial Intelligence*, 35:81–125, 1988.

[12] William G. Griswold. *Program Restructuring as an Aid in Software Maintenance.* PhD thesis, University of Washington, 1991.

[13] Patrick A. V. Hall and Geoff R. Dowling. Approximate string matching. *Computing Surveys*, 12(4):381–402, December 1980.

[14] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[15] W. Lewis Johnson and Martin Feather. Building an evolution transformation library. In *Proceedings of the 12th International Conference on Software Engineering*, pages 238–247, 1990.

[16] Won Kim. *Introduction to Object-Oriented Databases.* MIT Press, 1990.

[17] Peter W. Madany. *An Object-Oriented Framework for Filesystems.* PhD thesis, University of Illinois at Urbana-Champaign, 1992. Also Technical Report No. UIUCDCS–R–92–1751, Department of Computer Science, University of Illinois at Urbana-Champaign.

[18] Jeff McKenna. A proposal for change management for smalltalk. *Smalltalk Report*, 1(5):1–3, 1991.

[19] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, 1988.

[20] William F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992. Also Technical Report No. UIUCDCS–R–92–1759, Department of Computer Science, University of Illinois at Urbana-Champaign.

[21] William F. Opdyke and Ralph E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Proceedings of Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, September 1990.

[22] D. Jason Penney and Jacob Stein. Class modification in the GemStone object-oriented dbms. In *Proceedings of OOPSLA '87*, 1987.

[23] Edward J. Rak. Two redesign tools for Smalltalk. Master's thesis, University of Illinois at Urbana-Champaign, 1990.

[24] Marc J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, December 1975.

[25] Vince Russo, Gary Johnston, and Roy H. Campbell. Process Management in Multiprocessor Operating Systems using Class Hierarchical Design. In *Proceedings of OOPSLA '88*, San Diego, Ca., September 1988.

[26] Vincent Russo and Roy H. Campbell. Virtual Memory and Backing Storage Management in Multiprocessor Operating Systems using Class Hierarchical Design. In *Submitted to OOPSLA '89*, 1989. Also available as University of Illinois Technical Report.

[27] David Sankoff and Joseph B. Kruskal. Macromolecular sequences. In *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison (D. Sankoff and J. Kruskal, eds)*, pages 45–53, 1983.

[28] Robert A. Wagner. Order-n correction for regular languages. *Communications of the ACM*, 17(5):265–268, 1974.

[29] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software.* Prentice-Hall, 1990.

[30] Rebecca J. Wirfs-Brock and Ralph E. Johnson. A survey of current research in object-oriented design. *Communications of the ACM*, September 1990.

[31] Jonathan Zweig and Ralph Johnson. Conduits: A communication abstraction in C++. In *Proceedings of the USENIX C++ Workshop*, pages 191–203, 1990.