Reusing Object-Oriented Designs

Ralph E. Johnson
Vincent F. Russo

May 13, 1991

Please send us your comments about the paper. In particular, tell us what bores you, what mystifies you, and what you think is wrong. If you want to reference this paper, it is University of Illinois tech report UIUCDCS 91-1696.

Abstract

Reusing the products of the software development process is an important way to reduce software costs and to make programmers and designers more efficient. Object-oriented programming permits the reuse of design as well as programs. This paper describes two techniques for reusing design and how these reusable designs are developed. *Abstract classes* are reusable designs for components, while *frameworks* are reusable designs for entire applications or subsystems. These two techniques are related since frameworks almost always contain abstract classes. Although the most widely used frameworks are for user interfaces, this paper draws its examples from a framework for the virtual memory subsystem of an operating system.

1 Introduction

Experienced programmers reuse design. Therefore, a popular goal of software engineering is to develop tools and techniques to assist design reuse. The central problem with reusing design information is how to capture and express it [BR87]. Any design notation supports an abstraction that ignores some details of a problem and emphasizes others. This paper describes design techniques that emphasize objects and the interfaces between them.

Authors' address: Ralph Johnson, Department of Computer Science, University of Illinois, 1304 West Springfield Ave., Urbana IL 61801, Telephone: (217) 244-0093, e-mail: johnson@cs.uiuc.edu
Vincent F. Russo, Department of Computer Sciences, Purdue Univeristy, West Lafayette, IN 47907, Telephone: (317) 494-6008,, e-mail: russo@cs.purdue.edu

In particular, this paper describes two kinds of reusable object-oriented designs: *abstract classes* and *frameworks*. An abstract class is an incompletely specified class that is designed to be a template for subclasses, rather than a template for objects [GR83, WJ90]. An abstract class is a relatively small-scale design. A framework is a larger-scale design. It describes how a program is decomposed into a set of interacting objects. A framework is used to represent an entire application or subsystem [Deu89, WJ90]. Abstract classes are well-known inside the object-oriented programming community, but frameworks are not. Neither are yet understood in the wider software engineering community as an important way to reuse design.

In a recent paper, Mary Shaw argues that "software engineering" will not be a true engineering discipline until it codifies a large body of design information and creates reference materials that engineers can use to solve routine design problems quickly and reliably [Sha90]. We believe that frameworks are one of the most promising approaches for making software engineering a reality. The purpose of this paper is to show how object-oriented frameworks can be used to codify design knowledge for a particular application domain.

Since frameworks describe large-scale designs, any example will be large. Since frameworks codify design knowledge for a particular application domain, understanding a framework always requires understanding a little about its application domain. The main example of this paper is taken from the virtual memory system of the *Choices* operating system [RC89, Rus90]. *Choices* is a framework for operating system construction being developed at the University of Illinois.

The rest of this paper is presented as follows. Section 2 defines abstract classes and frameworks. Section 3 gives some background on virtual memory. Sections 4-6 describe the *Choices* virtual memory framework. Section 7 relates some of our experiences with the framework. The paper concludes by describing the process of developing frameworks.

2 Object-Oriented Programming and Reuse

The programming language features that characterize object-oriented programming languages are well-known and have been widely discussed. These features are data abstraction, polymorphism caused by late-binding of procedure calls, and inheritance. Although these features make it easier to reuse software, the major reason that object-oriented systems have been so successful at software reuse is a change in the way systems are designed. Design is reused, not just code, and a mature design is considered to be one that is easy to reuse and customize.

Data abstraction is an integral part of object-oriented programming. An object encapsulates both state and behavior. The only way to interact with an object is through its operations. In other words, the only way to determine an object's state is by its behavior. Usually the behavior (a set of operations) and the state associated with an object is defined by its *class*. A class is a template for the objects that are its instances.

The design-level view of a class differs from the implementation-level view by focusing entirely on the class's public operations. Public operations are those operations that are designed to be used by other objects. Thus, the design-level view of a class does not define state or private operations (those operations used solely by the class itself), although most of the public operations will eventually be implemented by accessing the state or by invoking private operations.

Polymorphism (the ability for a single variable or procedure parameter to take on values of several types) is another integral part of object-oriented programming. Polymorphism is achieved as a result of late-binding of procedure calls. The procedure to call in response to invoking an operation on an object is a function of the class of the object. Thus, a variable can take on the value of any object that implements the appropriate set of operations, *i.e.,* the variable can take on objects of several classes. Polymorphism makes programs more reusable and reduces the number of different interfaces. The kind of polymorphism provided by object-oriented languages leads naturally to a subtype relationship between types[CW85].

Most object-oriented languages provide class inheritance. Class inheritance lets one class, a subclass, inherit all the attributes (*i.e.,* the operations and state) of another class, the superclass. Inheritance can have many uses—code reuse, type checking, and categorizing components. This paper concentrates on its role in reusing designs (in abstract classes) and defining interfaces (in frameworks).

2.1 Abstract Classes

Abstract classes, the first of the two design techniques described in this paper, illustrate the power of inheritance to express designs. An abstract class is a class with at least one operation left unimplemented. Because some operations are unimplemented, an abstract class has no instances and is used only as a superclass. Thus, it is designed to be used as a template for specifying subclasses rather than objects. Although an abstract class lacks implementations for some of its operations, it can implement other operations in terms of the unimplemented operations. Classes that are not abstract are *concrete* classes. A concrete subclass of an abstract class will provide an implementation for any operation that needs one, and will inherit the implementations of the other operations.

For example, a File is an object with read, write, and size operations. File could have subclasses like UnixFile and NetworkedFile. Different kinds of files will have completely different algorithms for reading and writing the files, so class File will not implement these operations. However, these operations can be used to implement other operations. If the size operation gives the number of blocks in a file then any file can copy itself to another file using the copy operation, implemented as follows:

```
File::copy( File aFile ) {
  char buffer[BlockSize];
  for (int i=0; i++; i<= this -> size()) {
    this -> read(buffer);
    aFile -> write(buffer);
  }
}
```

A subclass of File that defines read, write and size will be able to use copy. Assuming that a subclass UnixFile of File does not redefine copy, copying a UnixFile to a NetworkedFile will invoke the read and size operations on the UnixFile and the write operation on the NetworkedFile. Thus, a function in a superclass can call a function implemented in a subclass.

Abstract classes have three kinds of operations. The read operation is an example of an *abstract operation*, which is not implemented by the abstract class but is left to subclasses to define. The abstract class provides a specification that all subclasses are to follow. Statically typed object-oriented programming languages check the syntactic part of the specifications, though a complete specification includes behavioral constraints like pre and postconditions and class invariants. Eiffel[Mey88] is the only commercial object-oriented language that provides any support for recording or checking the behavioral part of the specifications[1], and dynamically typed languages like Smalltalk do not even check the syntactic part.

The copy operation is an example of a *template operation*, which is an abstract algorithm defined in terms of one or more abstract operations. The subclass specializes the template operations by implementing the abstract operations. Thus, template operations are partially implemented.

A *base operation* is one that is fully implemented. For example, every file might have a variable that describes its owner, with functions owner and

[1] Eiffel provides pre and post conditions and class invariants that can be checked at run time. There are no tools for checking them statically, so their main use is for testing and documentation.

setOwner to access it. While base operations are useful, the abstract operations and template operations are usually more important parts of a design.

It is important to recall that abstract classes are a design technique, not a programming language feature. Most object-oriented programming languages provide no direct support for abstract classes. Programmers must rely on the documentation or a careful reading of the program (reverse engineering) to determine whether a class is abstract. For example, Smalltalk programmers indicate abstract operations by giving them an implementation that invokes the subclassResponsibility operation. However, Smalltalk performs little static analysis of programs, so it is easy for programmers to omit this important information. Early versions of C++ had no support for abstract classes, but recent versions allow abstract operations to be declared as "pure virtual"[ES90], and Eiffel lets them be declared as "deferred"[Mey88].

2.2 Frameworks

An abstract class is a design for a single object. A framework is the design of a *set* of objects that collaborate to carry out a set of responsibilites. Thus, frameworks are larger scale designs than abstract classes. Peter Deutsch emphasizes that the most important part of a framework is the part that describes how a system is divided into its components[Deu87, Deu89]. Frameworks also reuse implementation, but that is less important than reuse of the internal interfaces of a system and the way that its functions are divided amoung its components. This high-level design is the main intellectual content of software and is far more difficult to create or re-create than code. Frameworks are a way to reuse this high-level design.

Frameworks are similar to other techniques for reusing high-level design, such as templates [VK89] or schemas [KRT89, LH87]. Like abstract classes, frameworks are expressed in a programming language, but the other ways of reusing high-level design usually depend on a special purpose design notation. As a result, these other techniques have the potential to be more flexible and powerful than frameworks. On the other hand, they are still under development, while there are several successful and widely used frameworks.

Frameworks usually provide the design of only part of a program, such as its user interface, though application specific frameworks sometimes describe an entire program. We use the term *ensemble* to refer to an instantiation of a framework, *i.e.,* to a set of objects working together to carry out a set of responsibilities. An object-oriented system is (by definition) composed of a set of objects. These objects are only an ensemble if they follow the pattern decreed by a framework. A framework describes the architecture of an ensemble; the kinds of objects in the ensemble and how they interact. It describes how a particular kind of program, such as a user interface or network communication software, is decomposed into objects. It is represented by a set of classes (usually abstract), one for each kind of object.

Like abstract classes, frameworks reuse both design and code. Some aspects of a design, such as the kinds of objects, are easily described by code. Other aspects are not described well by code, such as invariants maintained by objects in an ensemble. The fact that some aspects of a framework are not expressed well as code makes frameworks harder to understand than abstract classes.

Any design notation will emphasize some details of a program at the expense of others. A framework concentrates on describing the objects that make up the program and how they interact. Dataflow is deemphasized, but communication paths between objects are emphasized. Although frameworks provide abstract algorithms, the users of a framework usually ignore the details of these algorithms and concentrate on designing and combining objects.

The concept of a framework is harder to understand than that of an abstract class. Abstract classes can be defined either by how to recognize them (*i.e.,* "a class that has some operations unimplemented") or how to use them (*i.e.,* "a class that is a template for subclasses, not objects"). Unfortunately, there is no simple way to tell whether a set of classes is a framework, so frameworks tend to be defined by how they are used. We will first define frameworks by example, and will finish this section with a more precise definition.

The first widely used framework, developed around 1980, was the Smalltalk-80 user interface framework called Model/View/Controller (MVC) [Gol84, KP88, LP91]. MVC showed that object-oriented programming was well-suited for implementing graphical user interfaces. It divides a user interface into three kinds of components; models, views and controllers. These objects work in trios consisting of a view and controller interacting with a model. A model is an application object, and is supposed to be independent of the user interface. A view manages a region of the display and keeps it consistent with the state of the model. A controller converts user events (mouse movements and key presses) into operations on its model and view. For example, controllers implement scrolling and menus. Views can be nested to form complex user interfaces. Nested views are called subviews.

Figure 1 shows a picture of the user interface of one of the standard tools in the Smalltalk-80 environment, the file tool. The file tool has three subviews. The top subview holds a string that is a pattern that matches a set of files, the middle subview displays the list of files that match the pattern, and the bottom subview displays the selected file. All three subviews have the same model—a FileBrowser. The top and bottom subviews are instances of TextView, while the middle subview is an instance of SelectionInListView. As shown by Figure 2, all three views are subviews of a StandardSystemView. Each of the four views has

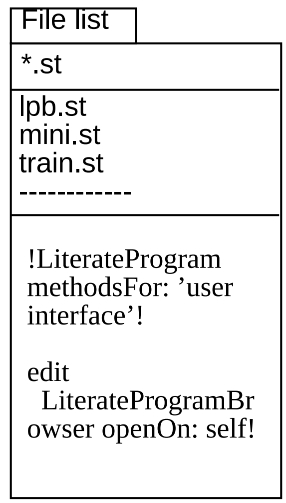

Figure 1: The Smalltalk-80 File Tool

its own controller.

Class View is an abstract class with base operations for creating and accessing the subview hierarchy, transforming from view coordinates to screen coordinates, and keeping track of its region on the display. It has abstract and template operations for displaying, since different kinds of views require different display algorithms. TextView, SelectionInListView, and StandardSystemView are concrete subclasses of View that each have a unique display algorithm.

As a user moves the mouse from one subview to another, controllers are activated and deactivated so that the active controller is always the controller of the view managing the region of the display that contains the cursor. Class Controller implements the protocol that ensures this, so a subclass of Controller automatically inherits the ability to cooperate with other controllers.

Class Object provides a dependency mechanism that views can use to detect when the model's state changes. Thus, any object can be a model. Later versions of Smalltalk-80 have added a Model class that provides a more efficient version of the dependency mechanism[PS88].

The file tool is a typical Model/View/Controller application that does not need new subclasses of View or Controller. The ensemble that makes up its user interface consists entirely of objects from classes that are a standard part of the Smalltalk-80 class library. The Smalltalk-80 class library contains several dozen concrete subclasses of View and Controller. However, when these are not sufficient, new subclasses can be built to extend the user interface.

Model/View/Controller has spawned many other user interface frameworks. MacApp is a popular commercial user interface framework designed specifically

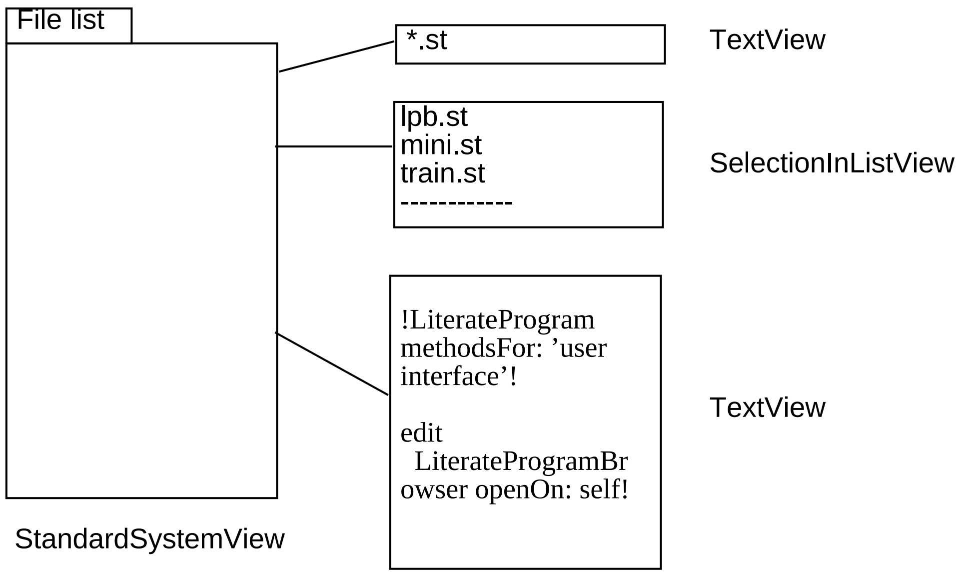

Figure 2: Subview Hierarchy in File Tool

for implementing Macintosh applications [Sch86]. Recently there have been several user interface frameworks from universities, such as the Andrew Toolkit from Carnegie Mellon University [PHK*88], InterViews from Stanford [LVC89], and ET++ from the University of Zurich [WGM88, WGM89]. Each of these frameworks improves the state of the art in user interface framework design in some way, building on the lessons of earlier systems.

Frameworks are not limited to user interfaces, but can be applied to any area of software design. They have been applied to VLSI routing algorithms [Gos90], to structured drawing editors [VL89, Vli90], code optimization [JM91], and psychophysiological experiments [Foo88]. Frameworks do not even require an object-oriented programming language. The Genesis database system compiler is a framework for database management systems [Bat88, BBR*89] as well as a tool for specifying how ensembles are configured in the framework [BB91]. Genesis does not use an object-oriented language but rather a macro processor and conditional compilation to implement an object-oriented design in C.

The important classes in a framework, such as Model, View, and Controller of Model/View/Controller, are usually abstract. Like Model/View/Controller, a framework usually comes with a *component library* that contains concrete subclasses of the classes in the framework. Like the file tool example, a framework is typically used by configuring or connecting objects from these predefined concrete classes. Ideally, an ensemble can be created entirely from classes in the component library. In practice, no component library is perfect or complete, and it is often necessary to derive new concrete subclasses of the abstract classes in the framework. Although a good component library is a valuable addition to a framework, the essence of a framework is not the component library, but the model of interaction and control flow among the objects of an ensemble.

Frameworks were informally defined earlier as a set of objects and how they interact. All the object-oriented frameworks mentioned in this paper are specified by informal documentation and a set of (usually) abstract classes that represent the objects. These classes give an operational specification of how objects in the framework interact. It would be better to formally describe the invariants that the objects are maintaining and the constraints on operation sequences, but the formal specification of object-oriented systems is not mature enough for that yet.

Given a specification language for describing the behavior of a set of objects, a framework would be a function from a set of objects to constraints on their behavior. These constraints would specify not only the syntactic interface of each object (*i.e.,* its type) but also the shared invariants and the legal operation sequences. For example, the Model/View/Controller framework would be a function from the three objects in a MVC trio

- to syntactic constraints like "the view has a function *image* from the state of the model to an image",
- to invariants like "the image on the screen inside a view's bounding box is its function *image* applied to the state of the model",
- and to operation sequences like "whenever the controller changes the state of the model, it performs the `changed` operation on it, which performs the `update` operation on the view, which will redraw the screen."

Note that a framework can define both an invariant and an outline of the algorithm for maintaining it. The message sequence above describes how Model/View/Controller ensures that the invariant above is maintained. We will see this intermingling of invariants and algorithms again in the framework for virtual memory.

Frameworks rely on undefined properties of the objects, such as the function from the state of the model to the image. A framework is filled out by selecting objects according to how they define these properties. The objects in a framework often place additional constraints on each other. For example, views (controllers) often use specialized operations to read (change) the state of a model.

The major problem with expressing frameworks in a programming language is that it is hard to learn the constraints that the framework imposes on its components by reading a program. The major advantage of expressing a framework as a program is that the algorithms and data structure of the program are automatically reused by every instantiation of the framework. There is usually an abstract class for each component in the framework. The algorithms and data structures that are reused are usually defined by these abstract classes. Each subclass of the abstract class defines a kind of component that fits into the framework, and they inherit much of their implementation from their abstract superclasses. This makes it much easier to develop a library of components that can be mixed and matched within the framework.

Ideally, a framework would be described both operationally and in terms of the constraints that it places on its components. This would not only provide code reuse, but make it easier to learn how objects in the framework interact. Work is being done to develop ways of describing the constraints formally [HHG90]. In the meantime, frameworks are being successfully described informally. The description of the *Choices* virtual memory framework in this paper shows how a framework can be described informally.

A framework reuses analysis, design, and code. It reuses analysis because it describes the kinds of objects that are important and how a large problem should be broken down into smaller problems. It reuses design because it contains abstract algorithms and describes the component interfaces that the programmer must implement and the constraints that any implementation must satisfy. It reuses code because it makes it easy to develop a library of compatible components and because the implementation of a new component can inherit most of its implementation from an abstract superclass. All of these are important, though in the long run it is probably the analysis and design reuse that provide the biggest payoff [BR87].

2.3 *Choices*

Choices is an operating system framework developed at the University of Illinois at Urbana-Champaign [CRJ87]. *Choices* was designed using the object-oriented paradigm and is implemented in an object-oriented language (C++). *Choices* is composed of interlocking frameworks for

- process management [RJC88],
- virtual memory management [RC89],
- file systems [MLRC88, MCRL89], and
- networking [ZJ90].

Over the last few decades, the number of computing environments has been growing rapidly. Often the needs of one class of system runs in direct opposition to another. For example, while AI programs demand large (virtual) address spaces, real-time embedded system constraints often lead to abandoning virtual memory altogether. Similarly, the memory management system for a hierarchical shared memory multiprocessor usually differs substantially from that for a small desktop workstation.

Choices is motivated by the realization that, despite all this diversity, computer users want to port applications from one environment to another, expect that what they learn about one operating system will be applicable to the next, and would like machines to interoperate.

One solution to the problem of conflicting requirements is to provide a *family* of operating systems that can be reconfigured to meet nearly any requirement. The need for a family of operating systems has been known for a long time [HFC76]. Just as automobile companies offer a wide selection of car sizes, engines, and colors to choose from, a family of operating systems would provide a set of components that could be rearranged in many configurations. Customers who want "drag racers" could combine "stock" components with a few customized components to build what they want. *Choices* is an attempt to construct such an operating system family.

Choices supports shared memory multiprocessors, real-time programming, a Unix-compatible programming environment, the ability to override almost any operating system function, and an object-oriented interface between applications and the operating system. Of course, it does not support all these features at once, since some of them are incompatible. Instead, it offers an array of features that can be selected on a per-operating system or, in some cases, per-user basis. In spite of this flexibility, the performance of *Choices* is similar to that of comparable operating systems [RMC90].

This paper describes the virtual memory framework of *Choices*, and uses it as an example of how object-oriented programming permits the reuse of design. The virtual memory framework provides an abstract design of a virtual memory system that can be customized to make a particular concrete virtual memory system. A framework hides the parts of the design that are common to all instances, and makes explicit the pieces that need to be customized. A framework is most valuable when the part of the design that is hidden is the part that programmers find hardest to understand. The virtual memory framework hides the details of how the operating system manages the sharing of physical memory among logical address spaces. In particular, it hides the details of synchronization among user processes and system processes managing memory, so programmers can be much less concerned about causing deadlock or interference. Hiding these concerns lets programmers deal with virtual memory more abstractly, which in turn makes it easier to experiment with virtual memory.

3 Virtual Memory

Understanding any framework requires understanding the problem domain for which that framework is designed. For the reader unfamiliar with virtual memory, this section will give a quick overview. Others may wish to skip this section. Those wishing more detail should refer to a suitable operating systems text [PS85, BS88, Tan87].

Most computers support the distinction between the memory referenced by a program and the physical (real) memory of the computer. Each program is given a *logical address space* in which to execute. Separate logical address spaces are used to protect one program from another, and to protect the operating system from malicious or aberrant programs.

Although a number of techniques have been used in the past, *paging* is currently the most popular way to provide separate logical address spaces. Paging divides physical memory into identically sized units or *frames* and logical memory into similar units called *pages*. The pages of a logical address space are arbitrarily mapped to physical memory frames.

Paging requires some sort of hardware support for the translation of logical addresses (the addresses generated by a program) to physical addresses (the addresses used by the hardware). Each time a processor accesses memory to fetch an instruction or data, the logical address is translated to a physical address by the *address translation unit* (see Figure 3). The precise implementation of the translation function used by the address translation unit is hardware dependent, so the details differ widely from one machine to another. However, the translation function is usually implemented by some form of "page table" indexed by the page number.

Since each program has its own address space, each has its own translation function and, therefore, its own table. The address translation unit is switched between these tables as different programs are run. Different tables for different programs will map the same logical addresses (pages) in different address spaces to different physical addresses (frames).

Paging simplifies protection and sharing of memory as well. By placing access rights in the page table, the hardware can check if a process is allowed certain access to an address during translation. For example, a *read-only* bit in a page table could prevent a program from accidentally writing over its code. Sharing can be provided by having the page tables for different programs map pages in different logical address spaces to the same physical memory frames.

Paging makes it easy to relocate programs in a multiprogramming environment. It does not necessarily allow programs to be larger than the amount of physical memory available on the computer. *Virtual memory* is a logical memory technique that provides the illusion of having more memory than is

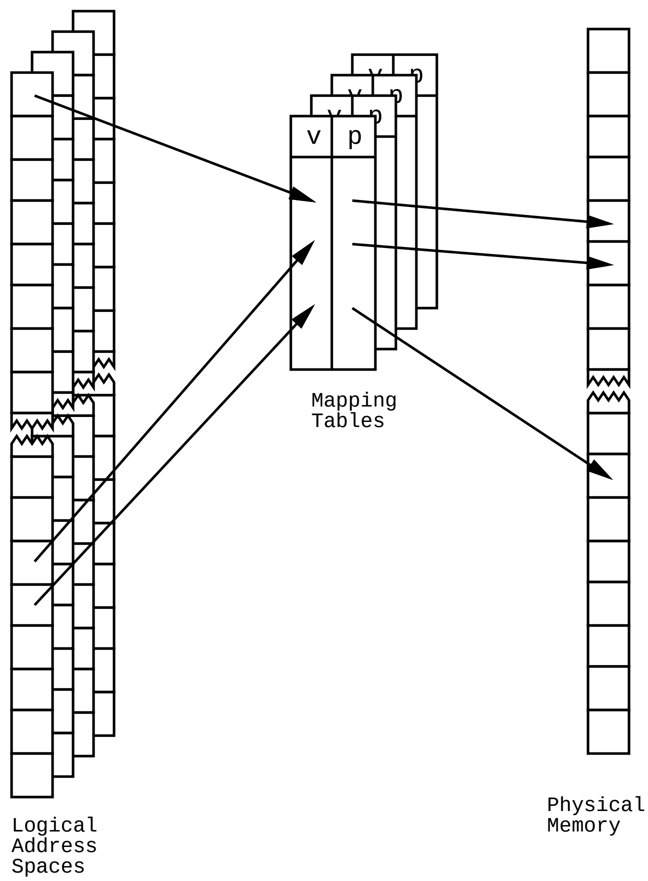

Figure 3: Paging and Address Translation

actually on the computer by using physical memory as a cache for data on (much larger) disk or other secondary storage. The page table describes what is in the physical memory cache of virtual memory and where in that cache it can be found. The only thing necessary to support virtual memory is the ability for the page table to trap access to an invalid page. This is usually accomplished though a *valid-bit* associated with each page that indicates whether the physical memory cache contains the data for that page.

A *page fault* occurs when physical memory does not contain the data addressed by a program (the valid-bit is not set), or when the attempted access does not match that allowed by the page table (for example, writing a read-only address). The address translation unit detects a page fault while translating a logical address to a physical address and interrupts the processor. A page fault interrupt is handled by the operating system. If the fault was caused by an invalid access then an appropriate exception handling routing is invoked. If the page fault was caused by a "cache miss" (physical memory does not contain the requested data) then the operating system brings the missing data into memory, updates the page table, and restarts or resumes the operation that caused the page fault. If physical memory is full of data already being cached then the operating system must first free space by moving some data back to secondary storage and updating page tables to cause new page faults if those data are later accessed. This is usually called *page replacement*.

4 The *Choices* Virtual Memory Framework

The problem of operating system support for virtual memory can be divided into two sub-problems: representing the logical data to be cached in physical memory, and maintaining this caching. The first problem is traditionally the file system's responsibility rather than the virtual memory system's, but the notion of persistent data is common to both.

Paralleling these two sub-problems, an application programmer's view of the *Choices* virtual memory system is based on two kinds of objects: *memory objects* and *domains*. A *memory object* represents a set of data, such as a disk or file. A *domain* represents a complete logical (virtual) address space. In the *Choices* model, a virtual address space maps virtual addresses to data in memory objects. Each memory object is assigned a range of addresses within the total address space. Addresses that fall within a given range refer to data in the corresponding memory object. Domains encapsulate the function of caching logical data in physical memory. They abstract page tables and physical memory management, hide the details of page faults, and ensure consistency of information about the location of data.

The *Choices* virtual memory model is similiar in design to that of other

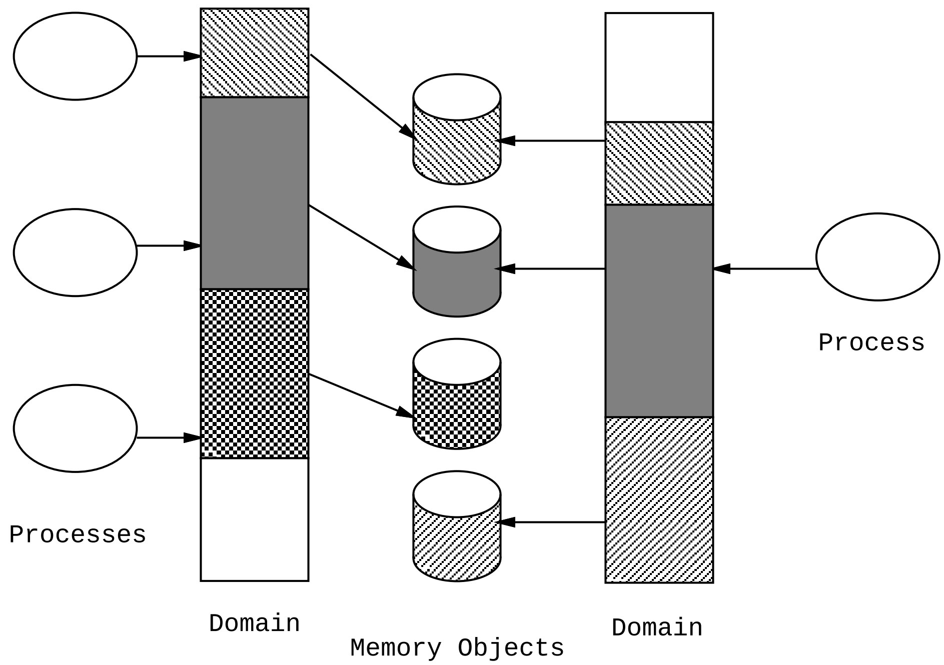

Figure 4: The Choices Virtual Memory Model

modern operating systems including Mach[RTY*87], CHORUS[RAN88], and V[Che88]. Figure 4 shows an application programmer's view of the *Choices* virtual memory system. Each domain has one or more processes that execute within it. A memory object can belong to any number of domains. A process can have its own private address space, share an address space with other processes, or share part of its address space (in the form of shared memory objects) with other processes.

MemoryObject

All logical storage abstractions are subclasses of the abstract class MemoryObject, including physical memory, disks, and files. A memory object is an array of logical "units", which are equal-sized fixed length blocks of bytes. Units correspond to disk blocks or physical memory frames, although it is often useful to have the unit size of a memory object be different from the block size of the underlying hardware. A memory object implements four operations, read, write, numberOfUnits, and sizeOfUnits. These allow the user to read and write a number of units from/to the memory object and learn the characteristics of the unit array. The numberOfUnits operation returns the length of the array, and the sizeOfUnits operation returns the number of bytes in each unit.

The file system of *Choices* also uses the MemoryObject class hierarchy extensively. It might seem odd that an important part of the virtual memory framework is also important in other parts of an operating system, but one of the main advantages of object-oriented programming is factoring out commonality. Both the file system and virtual memory system rely on the read and write operations of MemoryObject. The virtual memory system uses them to obtain data to fill the cache, while the file system uses them to fulfill higher level file read and write requests. Reusing an abstraction from the virtual memory system within the file system is an example of how well-designed abstractions can simplify a large system.

Domain

A Domain maps a virtual address space onto a set of non-overlapping memory objects. A typical process will access several memory objects. There will be one for its program, one for its local variables, one for its shared variables, one for its stack, and one for each file that it references. The process's domain maps the process's address space onto these memory objects. A domain also associates a protection with each memory object. For example, a program memory object could be read-only and shared by other domains, while the local variable memory object is writable and private and the shared variable memory object is writable and shared.

A domain can ensure that its memory objects are non-overlapping because it is the domain, not the memory object, that determines the location of the memory object in the virtual address space. This is important because it allows memory objects to be moved around in the domain without being notified. Since memory objects can grow, it might be necessary to move them to ensure that their address ranges do not overlap.[2]

Domains have operations to add and remove memory objects (add, remove), to find the memory object at a particular logical address (lookUp), and to handle a page fault (repairFault). Each domain contains a page table and updates the page table as memory objects are removed and as data are read in to physical memory in response to a page fault and written out for page replacement.

Design Issues

A domain and a memory object are both complex objects, but they are complex in different ways. A domain has a well-defined structure with several kinds of components, which implement different virtual memory policies and different

[2]Memory objects can be moved only if they contain position independent (relocatable) information. Text files are a good example.

kinds of address translation units. Domains are not used as building blocks to make larger objects, but are always top-level objects. Moreover, class Domain is concrete and has never been subclassed.

In contrast, memory objects have less structure than domains, but can be used as building blocks to make high-level memory objects. MemoryObject is an abstract class with many subclasses. Instances of some of the subclasses modify other memory objects. Since there are so many kinds of memory objects, a "memory object" is more of an interface than an implementation.

The next two sections describe the structure of memory objects and domains in more detail. Domain and MemoryObject represent two different ways that object-oriented programming allows the reuse of design. MemoryObject is a typical abstract class, while Domain illustrates how a framework describes how an ensemble is customized by replacing its components.

5 Memory Objects

Memory objects form the foundation of the *Choices* virtual memory and permanent storage management framework. Some memory objects access disks or physical memory directly, but most are complex ensembles of other simpler memory objects.

Much of the customization of *Choices* is done by combining existing kinds of memory objects to make new ensembles. It is unlikely that a finite set of classes will be able to efficiently describe every kind of memory object that could ever be needed. Inheritance, however, makes it easy to extend the framework by adding a new class of memory object to the component library as the need arises. The rules the framework defines for connecting together new and existing classes into new ensembles further aids in customization. For example, a domain allows any new memory object ensembles to be used as a backing storage.

5.1 Disk

Class Disk is a primitive memory object that is equivalent to a device driver in a conventional operating system. A Disk is rarely used directly. Rather, it is usually referred to by some other memory object that uses it for storage. Even though there might be multiple indirections involved, Disks form the eventual sources and sinks of almost all data in *Choices*. The read and write operations of Disk read and write physical blocks on the disk. The sizeOfUnits operation gives the size of a disk block, and the numberOfUnits operation returns the number of blocks on the disk.

To make *Choices* more portable, Disk is an abstract class. Different kinds of hardware disk interfaces require different subclasses of Disk. Each subclass encapsulates the device controller, interrupt management mechanism, *etc.*, for a particular computer that *Choices* is targeted to. A framework for constructing new Disk classes is currently being designed[Kou91].

5.2 MemoryObjectView

To be useful, large memory objects like disks must be partitioned into smaller memory objects like files. Class MemoryObjectView provides a view of a part of a larger memory object. Since most memory objects simply provide different interpretations of underlying memory objects, MemoryObjectView is the superclass of most memory object classes in the *Choices* component library. A MemoryObjectView must know the memory object on which it is stored and must know how to translate from a block number in the view to a block number in the memory object on which its data are stored. The read and write operations are easy to implement and do not perform any actual I/O. They simply translate the requested block number and forward the operation to the memory object on which the file is stored.

There are two primary subclasses of memory object view. ContiguousView provides a view of a contiguous subrange of the underlying memory object. Thus, in addition to a reference to its source, it needs two pieces of information about its location, the starting block and the size. Its n'th block can be found by adding n to the number of the starting block. ContiguousView is a concrete class and has never been subclassed. Its most frequent use is in the construction of new memory object ensembles (see Section 5.3).

The other primary subclass of MemoryObjectView is DiscontiguousView. DiscontiguousView is an abstract class specifying a set of units from the memory object being viewed, but not a continguous set. Concrete subclasses provide a function that maps from the unit numbers of the DiscontiguousView to the underlying memory object. For example, a Unix file uses i-node tables and indirect blocks to map logical block numbers (blocks in the view) to underlying physical block numbers (blocks in the source). It will often read several blocks from its memory object before it can find the data block to read or write. For a further discussion on how the *Choices* memory object hierarchy was used to implement several versions of the Unix file system, see[MCRL89].

5.3 Customizability

The simplest and most common way to customize the *Choices* virtual memory system is by creating new kinds of memory objects. Inheritance makes programming new classes easier, but polymorphism avoids the creation of new classes by letting new memory object ensembles be built from existing classes. Most uses of memory objects, including those in the implementation of Domain and MemoryObjectView, depend only on the memory object interface and not on any particular subclass. This means that any memory object can be placed in the address space of a program or viewed by a MemoryObjectView.

One possible application of this polymorphism is the ability to have many different file systems, each with its own format. In fact, it is possible to format a file like a foreign file system and use it with software that was written for the foreign file system. Likewise, since the source of a MemoryObjectView is simply another MemoryObject, any memory object can be broken into files, not just disks. Finally, networked file systems, special file systems for database management systems, even special archival file systems that compress the component files, can all be implemented as classes derived from MemoryObjectView and treated like any other memory object.

The virtual memory system can be customized even without creating new classes. For example, Unix uses a special contiguous swapping area on the disk as backing store for each process, which makes paging a lot faster than if the file system were used. The same effect can be achieved in *Choices* by using a memory object mapped to a contiguous region of the disk (a ContiguousView). Thus, the *Choices* virtual memory system can mimic this aspect of Unix without creating new classes. On the other hand, it is also easy to page to a user file, since a domain can use any memory object.

It is important to distinguish between reuse by inheritance and reuse by connecting objects together to form a new ensemble. The top half of Figure 5 shows part of the memory object class hierarchy. The concrete leaf classes inherit the implementation of their abstract superclasses. In contrast, the bottom half of Figure 5 shows how complex memory objects can be built from existing classes using polymorphism. It shows the memory object ensemble representing the executable part of a compiled program. The base of this ensemble is the object (an instance of MultimaxDisk) representing the disk. The disk is broken into partitions by a ContiguousView. One of these partitions is further decomposed into files by an object in BSDInode, which is a subclass of DiscontiguousView. This memory object corresponds to an `a.out` file in a Unix system. An executable program file consists of several parts, including not only the program text but also initialized data and symbol tables. The program text is a contiguous subsection of the file, and so can be selected by another ContiguousView. Complex memory objects are implemented as an ensemble of simpler memory objects, not by inheritance, so it is easy for them to share the state of the disk or of an intermediate memory object.

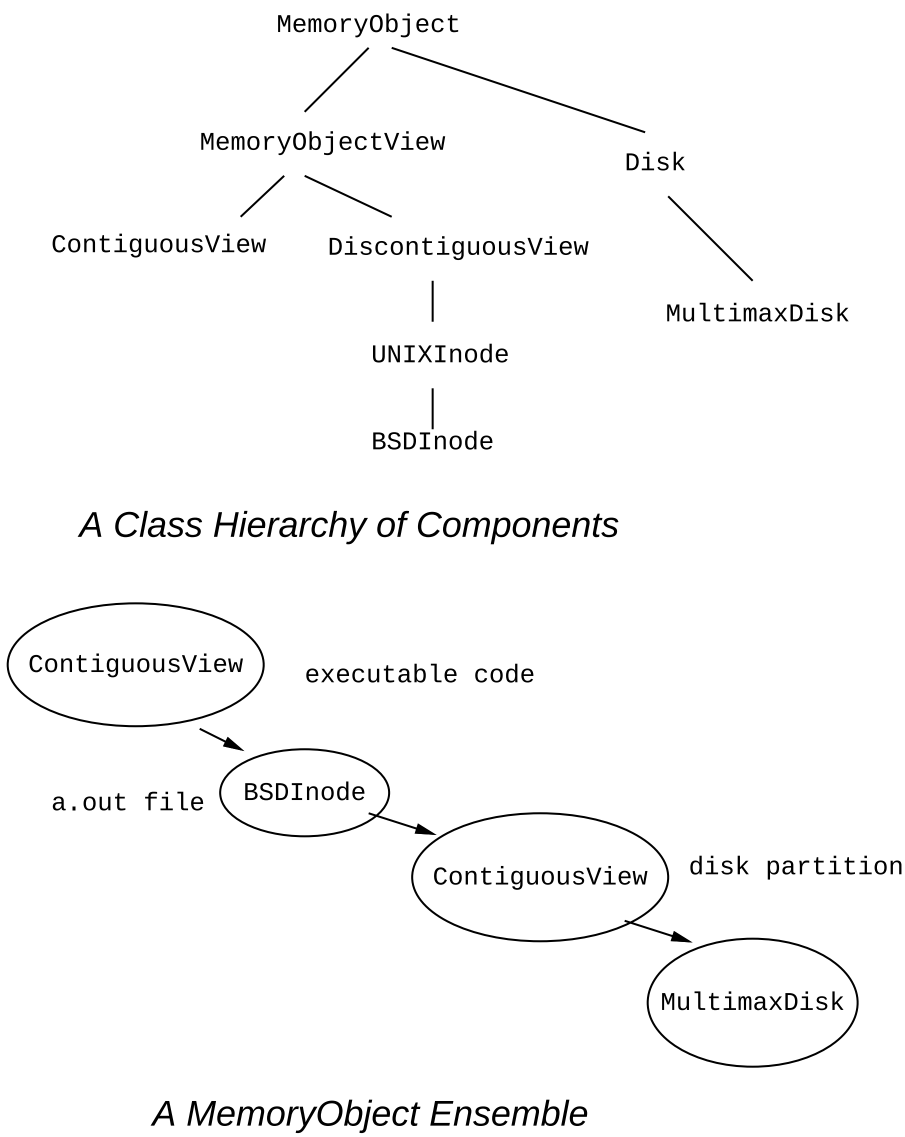

Figure 5: A Memory Object Instance Hierarchy

6 Domains

The design of class Domain is quite different from that of MemoryObject. A Domain delegates to its components the responsibility for managing a page table, implementing the paging policies, and allocating physical memory. These components are instances of subclasses of AddressTranslation, MemoryObject-Cache, and PhysicalMemoryManager, respectively. AddressTranslation and MemoryObjectCache are both abstract classes. Each of these classes is the root of a class hierarchy of components. Domain is concrete. A programmer builds a customized domain by changing its components, not by subclassing Domain.

The design of class Domain is complicated by the fact that a memory object can be shared by several domains. Each memory object must keep track of the physical addresses where its pages are cached in primary memory. This is because a unit of a memory object should never be cached in physical memory more than once. If a Domain kept this physical address information, it would have to coordinate it with all other Domains.

MemoryObjectCache keeps track of which units of a memory object are stored in physical memory (*i.e.,* are cached) and where. This is machine-independent information. AddressTranslation, on the other hand, represents the machine-*dependent* page table that the address translation unit uses. A set of MemoryObjectCaches in a domain and the AddressTranslation for that Domain contain redundant information; the MemoryObjectCaches represent a mapping from logical to physical addresses in a way that is easy for a programmer to manipulate while an AddressTranslation represents it in a way that is possible for the hardware to use.

One of the invariants of the virtual memory framework is that the mapping of the AddressTranslation is always a subset of the mapping of the set of MemoryObjectCaches. Another invariant is that a domain is a map of an address space to a non-overlapping set of memory objects. Likewise, the one-to-one mappings of caches to memory objects and domains to address translations are invariant. The interactions of all the *Choices* virtual memory components are shown graphically in Figure 6.

6.1 Memory Object Caches

A MemoryObjectCache maps the data of a memory object into physical memory. It can cache all, part or none of a memory object's data, and keeps track of the physical address of each unit that has been cached.

The main operations on a cache are cache, release, and protect. The cache operation ensures that a particular unit is in the cache, while the release operation removes a unit from the cache. The cache operation always returns

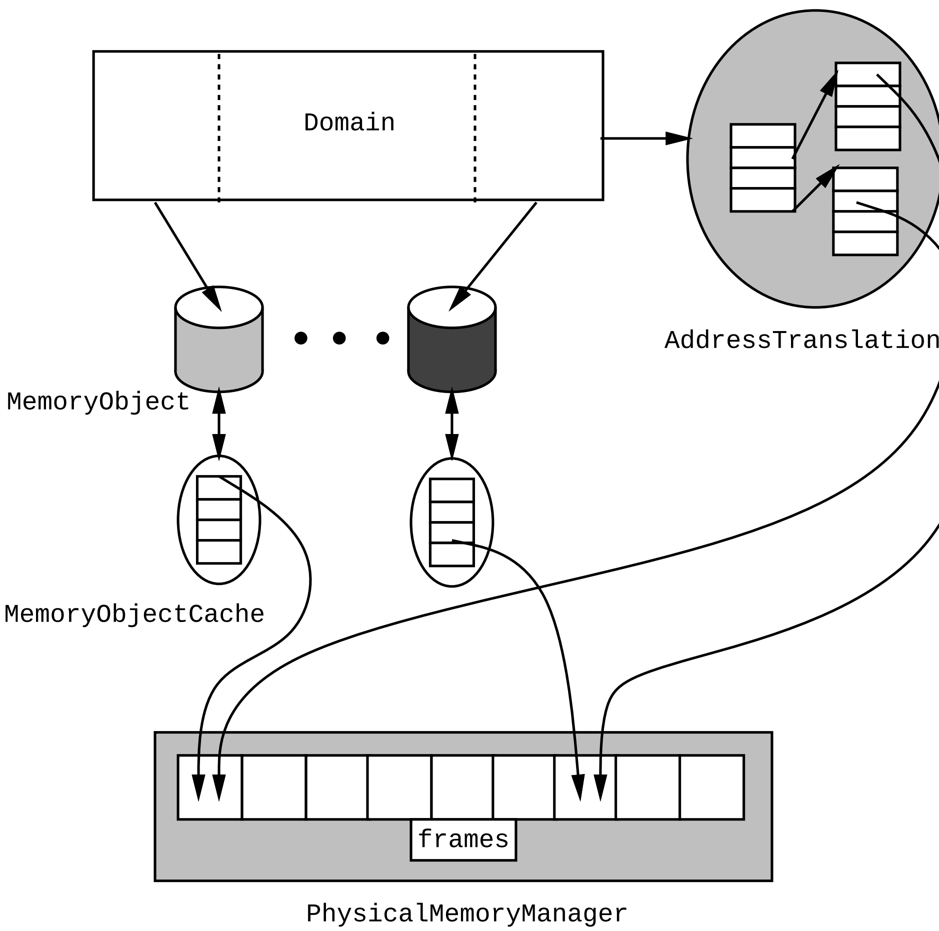

Figure 6: An Overview of the Virtual Memory Framework

the physical address of the unit, even if the unit was already cached, so it can be used to find the physical address of a previously cached unit. Each unit is given a protection level when it is cached; protect sets the maximum protection level of a unit, and can change the protection of an already cached unit.

These are all abstract operations. The most commonly used concrete subclass is PagedMemoryObjectCache, which implements cache by allocating physical memory for the unit and reads the unit from the memory object in page frame sized quantities. Similarly, release removes a unit from the cache by deallocating the physical memory and writing the unit back to its memory object if it has been changed.

MemoryObjectCache has a few template operations like releaseAll and protectAll. These are never redefined by subclasses and are not as important as the three main abstract operations.

Different kinds of caches implement cache in different ways. A conventional cache will obtain physical memory, read a unit of data from disk and put it into that memory, and update its internal mappings to show that the data is now present. Thus, caches must be able to communicate with a physical memory manager (a PhysicalMemoryManager), and a disk or backing store (a MemoryObject). Section 7 will describe how some other subclasses of MemoryObjectCache implement cache.

6.2 Address Translations

Each domain has its own page table. Although the page table is logically part of the domain, its format is machine dependent. To keep the implementation of Domain host architecture independent, the page table has been split from Domain into a separate object of class AddressTranslation.

The abstract operations of AddressTranslation are addMapping, removeMapping, and changePermission. These correspond to the three abstract operations of MemoryObjectCache, since putting a unit in the cache requires adding a mapping for its location in physical memory, releasing a unit from the cache requires removing its mapping, and changing the protection level of a unit in a cache is the same as changing the permission in an AddressTranslation (a page table). Note that only domains communicate directly with page tables, so caches return the physical memory location when they cache a unit and let the domain update the page table.

A physical address is represented by an object of class Frame. Thus, a page table is a mapping from virtual addresses to frames, and a cache is a mapping from unit numbers to frames. A frame is only used by one cache at a time, but it will be used by more than one page table if its memory object is shared by several domains. It keeps track of the page table and the cache that uses it.

This makes it easy to notify all page tables that use a frame when the frame is removed from its cache, *i.e.,*when it is deallocated.

6.3 Physical Memory Manager

A physical memory manager (an instance of class PhysicalMemoryManager) has two public operations, allocate and free. Caches use these operations to acquire and dispose of physical memory (frames).

The major design decision for a physical memory manager is what to do when there is no physical memory to allocate. The current design relies on the fact that there is a process that periodically releases units from caches, thus deallocating physical memory. This process ensures that there is always physical memory available. Thus, the physical memory manager blocks when it runs out of physical memory.

PhysicalMemoryManager is a concrete class and has never been subclassed. It would be easy to make it abstract and it is easy to imagine useful subclasses. Subclasses could provide more robust handling of running out of physical memory. They could also manage hierarchical physical memory. However, we have not needed any of these features yet, and so have not yet made PhysicalMemoryManager abstract.

6.4 Abstract Virtual Memory Algorithms

An important part of most frameworks are the abstract algorithms that define operation sequences. The most complex abstract algorithm in the virtual memory framework is the one for responding to page faults. A page fault occurs whenever the computer accesses an address that is not mapped by the page table. The page fault handler (an Exception[CRJ87]) will perform a repairFault operation on the current domain. The repairFault operation will determine the cache corresponding to the address at which the page fault occurred and will then ask that cache to cache the data (using cache), giving it information about the kind of access that faulted (read, write, execute) and the address relative to the start of the cache. The cache operation returns the physical address of the data, so the domain can update its page table (*i.e.,* the AddressTranslation). The domain then returns to the fault handler, which will restart the instruction.

In addition to repairFault, Domain implements add, remove, and locate. Each domain has an array of memory objects and their starting addresses. locate returns the memory object managing particular address along with the offset into that memory object. The add method adds a memory object to the array. The add operation does not have to update the page table because it will be updated by a page fault the first time that an address is accessed that is assigned to the memory object. However, the remove operation, which deletes the binding of an address range to a memory object, must update the page table to remove all mappings belonging to the memory object.

Note that these abstract algorithms differ from template methods in that they can be customized by changing the components of the domain, while template methods are customized by creating a new class.

6.5 Customizing Domains

Domains are customized by changing their components. The two main ways that they are customized is by changing AddressTranslation (to port the virtual memory system to a new machine) and by making new kinds of caches. Most new caches are subclasses of PagedMemoryObjectCache, which is a concrete class that implements the standard virtual memory algorithms. The new cache classes reuse the standard virtual memory algorithms, but usually extend the cache function, as described in the next section. Domain and PhysicalMemoryManager have not yet needed to be subclassed.

7 Applications of the VM Framework

Recently there have been a number of proposals for using address translation hardware to implement operating system and programming language features that traditionally are not part of the responsibility of the virtual memory system. These include proposals for using address translation hardware for garbage collection[AEL88, BDS91, Sha87] and for distributed programming[LH86]. If a feature is normally expensive because it requires checking some condition regularly, it can often be made inexpensive by translating the condition into a protection or validity check that can be performed by the address translation unit. Techniques that use address translation hardware have prompted interest in making virtual memory management systems more customizable[AL91].

Section 5 showed how the virtual memory framework can be customized using memory objects. Section 6 discussed the details of how the virtual memory framework is put together and showed how AddressTranslation makes the system portable. This section will describe several advanced applications of the *Choices* virtual memory framework. Two of these—distributed shared memory and garbage collection—are well-known, the third is new.

7.1 Distributed Shared Memory

Distributed shared memory is a way to let programs designed for shared memory work in a distributed computing system by emulating the shared memory across a network[LH86]. This is done by treating each page of data separately and automatically moving them from machine to machine as needed. When a process accesses a non-resident page, it causes a page fault. The page is then copied from another machine, the page tables on that machine invalidated, and the page tables on the first machine updated. When a page is only being read, it can have multiple copies on different machines. However, as soon as one machine attempts to write to the page, those copies must be invalidated and recopied from the newly written copy.

Distributed shared memory is currently implemented in *Choices* by DistributedMemoryObjectCache, a subclass of MemoryObjectCache, and by subclasses of PageRecord [JC89, SMC90]. A PageRecord is used to represent the state associated with a unit of a memory object, such as its current location, whether it is cached in physical memory, and whether it is being fetched from another processor.

In retrospect, it would probably be better to implement distributed shared memory by subclassing MemoryObject, not MemoryObjectCache. MemoryObject is simpler and easier to change. Moreover, caches deal mostly with physical memory, while memory objects represent logical memory. PageRecords primarily describe logical memory, and so are more naturally a part of a memory object.

A distributed memory object would be shared by many machines by having a local version of the object exist on each machine. Each local version would know about its peers on the other machines. Since it is a memory object like any other, when a page fault occurs, the distributed memory object will be requested to fetch the data. The difference is that a distributed memory object will, when the page is not resident locally, contact one of its peers on another machine and request the page be removed from that machine and transferred to the first machine. If the second machine later attempts to access those data, it will in turn fault and request the page back. If the memory object is shared by more than two machines then the other machines may be informed of the new location of the page so that they will know how to respond to a page fault as well.

7.2 Garbage Collection

Another interesting use of virtual memory hardware is for garbage collection. Appel, Ellis and Li proposed a real-time garbage collection algorithm that is fairly efficient, does not require special hardware, does not impose many restrictions on the mutator[3], and works well with multiprocessors [AEL88]. Earlier real-time garbage collection algorithms were either much less efficient

[3]The mutator is the application program allocating new cells and modifying existing ones.

or required special hardware support, so this algorithm has attracted a lot of interest.

Their algorithm is a copying garbage collector, which means that the garbage collector copies all the non-garbage cells from an old memory region to a new region, leaving the garbage behind. Incremental copying garbage collectors are usually inefficient because they require each access by the mutator of a cell to check whether the cell is in the old region or the new region. The new algorithm uses the address translation unit to check whether a cell is in the old region or the new region. The pages in the new region are divided into those pointing to cells in the new region and those pointing to cells in the old region. A page that points to cells in the old region can be made to point to cells in the new region by examining each cell in the old region. If the cell has been copied then it will have a pointer to its new location in the new region, and its old address can be replaced by its new one. If the cell has not been copied then it can be copied to the new region and a forwarding pointer can be left behind from its old location to its new one.

Mutators will only access pages in the new region. Pages that only point to pages in the new region are safe to access, but those that point to pages in the old region are not. The page table only permits access to safe pages, so accessing an unsafe page causes a page fault. The page fault will make the page safe by copying to the new region all objects in the old region to which the page points.

This garbage collection technique is especially well suited to a multiprocessor. It is not necessary to wait until a page is accessed to copy it. Instead, a processor can be assigned to convert unsafe pages into safe pages. Thus, the application program can have very little garbage collection overhead.

This algorithm has been implemented in *Choices* by defining a subclass of MemoryObjectCache called GarbageCollectableCache. A pair of GarbageCollectableCaches represents the old region and the new region. Only the new region is in the address space of the mutator. Initially all of the pages in the new region are unsafe, but each page fault invokes the cache operation of the GarbageCollectableCache, which makes the page safe. Eventually all the pages are safe, and garbage collection is over. Another class, GarbageCollectorManager, is responsible for keeping track of which cache is the new region and which is the old region, for initiating garbage collection, for managing the process that does garbage collection in the background, and for allocating pages for new objects. The mutator interacts directly only with the GarbageCollectorManager, though it interacts indirectly with the GarbageCollectableCache by causing page faults.

The current version of this algorithm in *Choices* is as simple as possible, and does not implement multiple generations, a separate space for large objects, or any other extra feature. GarbageCollectableCache has a little more than 100 lines of code and 45 lines of header file, while GarbageCollectorManager has about 40 lines of code and 20 lines of header file. There is an additional 170 lines of code and 80 lines of header file that is pure garbage collector, *i.e.,* it could be used in a conventional copying collector.

7.3 Futures

A *future* is a value in the process of being computed [Hal85]. It is a parallel programming construct that combines a process with a synchronization mechanism. A future can be assigned to a variable, passed as an argument to a procedure, and treated just like any other value. If a process needs the value of the future, it will block until the future's value has been computed. In order to achieve this, there is a lock associated with each future. Accessing the value of the future requires checking the lock and blocking the current process until the value of the future has been computed. Once the value of the future "arrives", the future is treated just like any other object, and all processes that are waiting for the future are unblocked. Likewise, a new process that requests the value of the future need not block.

Multilisp lets a program use futures anywhere it uses other kinds of values, and lets the programmer replace any value with a future [Hal85]. This makes futures easy to use but hard to implement efficiently. Each access of any object must determine whether the object is a future and, if so, whether its value has been computed. Thus, all object accesses are slow because of futures.

In contrast, a Concurrent Smalltalk future (called a cbox) is distinct from other values [YT86]. Its only operation is receive, which waits for its value to be computed and then returns the value. This eliminates the need for checking whether an object is a future by giving futures a unique interface, but prevents a procedure that was written for a sequential program from being used with futures.

The *Choices* virtual memory system makes it possible for futures to be used like any other kind of value and still be efficient [Lad89]. Futures are implemented by defining a subclass of MemoryObjectCache, FutureCache. A future cache contains a collection of futures, one for each page. Instead of filling the cache by reading from a memory object, a future cache fills itself by waiting for the future's process to finish. Thus, FutureCache redefines the cache operation.

A future of type T consists of space for an object of type T and a lock. Pages holding futures in the process of being computed are marked as invalid in the page table, so that any access of those futures causes a page fault. Processes that cause a page fault because they accessed a future block on the lock for that future. When a future's value is computed, all processes waiting for the future are unblocked and then pages holding the future are marked as valid in the page table.

Creating a future requires allocating its space and starting a process to compute its result. When the process is finished, it marks the pages holding the future as valid and unblocks the processes waiting for the future. Although each future is allocated a separate page in the FutureCache, it would waste physical memory to give each small future its own frame. Physical memory is more scarce than virtual memory, so small futures share physical memory frames and the future cache will map several pages to the same frame.

A future's process must be able to store into the future when its value is computed. On the other hand, a process using the future must trap if it tries to use the future. Both processes are user processes and are part of the same domain, so it is impossible to have one of them trap when it accesses the address of the future, but not the other. This is solved by giving the future two addresses, one that is writable and the other that is not. The future process is the only process given the writable address. The FutureCache ensures that the memory object that stores the futures can be accessed using either address.

This design makes futures flexible without making the entire system less efficient. Programs that use futures suffer a performance penalty only when they actually access an unfinished future, since there is no explicit check in the program for futures. Since the page containing the value is marked valid once it has been computed, there is no penalty at all to access a future's value a second time. Likewise if the value is never accessed until after it has been computed, there is no cost other than the overhead the future computation process imposes on the system. Likewise, programmers can use a pointer to a future anywhere they could use a pointer to any other object. No change to the compiler or run-time system is necessary to implement futures in *Choices*; it is entirely an operating system function.

FutureCache defines three operations for a total of 70 lines of C++, and was the only addition to the kernel. There are also several hundred lines of library routines that run in user space to create a future and to assign it a value.

7.4 Experience with the Virtual Memory Framework

The *Choices* project has received several benefits from frameworks. A common set of classes gives the project members a common vocabulary. The frameworks make it easier to try out new ideas, so research on operating system topics is more productive. *Choices* is easy enough to use that students have used it for class projects.

An important benefit of frameworks is that they help coordinate people working on the same project. One way that they do this is by providing standard interfaces that are widely reused. Thus, the different projects that reuse the virtual memory framework have all been compatible with each other because they have reused the interfaces provided by the framework.

A second way frameworks help coordinate a project is by providing a way to divide work. Not only can work be divided according to which framework it will use, but it can be divided into those improving, extending, or developing frameworks and those using them for a particular application. The *Choices* project has usually had a single person in charge of each framework, but each framework has many users. Although everyone has to use the virtual memory system, the real users of the virtual memory framework are those who have extended it (half a dozen) or ported it (another half a dozen). Each of the projects described in section 7 was done by someone other than the framework designer.

A use of a framework validates it when the use does not require any changes to the framework, and helps improve the framework when it points out weaknesses in it. For example, the implementation of futures did not require changing any existing classes, but just added one new class with three functions to the kernel. On the other hand, other projects required revising the framework. The garbage collector required several changes to existing classes, though these were mostly design oversights. Distributed shared memory required the most changes.

The *Choices* virtual memory framework, like all the other parts of *Choices*, has changed several times. Usually the virtual memory framework was reorganized to simplify it and to make it easier to understand and reuse. We could argue that any virtual memory feature can be added to *Choices* since, at worst, the implementation of Domain could be replaced. However, any feature that requires replacing Domain shows that the virtual memory framework has serious flaws. A good framework should be designed to be extended easily using polymorphism and inheritance. If a framework cannot be extended to solve a particular problem in its problem domain then it has failed.

Change was almost always motivated by trying to reuse the framework. It is only after a framework has been used several times that it is possible to tell what kinds of extensions are difficult. For example, experience porting *Choices* has led to a desire for a framework for device controllers to make it easier to implement primitive memory objects, like Disk. Similarly, the virtual memory applications described earlier imply that MemoryObjectCache needs to be easy to change, and suggest that we need to extend the framework to describe the state of each unit of the memory object.

A framework is improved by pushing on the boundaries of the problems that it solves best. *Choices* addresses the problems in the design of a conventional virtual memory system, but we haven't used it to provide user customizable paging (other than selecting from choices built into the kernel), implementing a virtual machine operating system, or several other features. Some of these applications will be harder to implement than we think they should be, which will cause us to improve the framework. Thus, the *Choices* virtual memory framework will continue to improve as it is used.

8 The Process of Framework Design

One of the most common observations about framework design is that iteration is essential [JF88, Wir90]. Batory *et. al.* emphasize that the development of their framework took much longer than expected and attributed this to the difficulty of domain analysis [BBR*89]. Our experience supports theirs.

The question remains: Why is iteration necessary? Clearly, a design is iterated only because its authors did not know how to do it right the first time. Perhaps this is the fault of the designers: they should have spent more time analyzing the problem domain, or they were not skilled enough. Lack of experience with the problem domain certainly contributed to the length of time that it took to design the *Choices* virtual memory framework. None of the people on the *Choices* project had ever developed a complete operating system, and most of them learned object-oriented design while working on the *Choices* project. However, lack of experience is not the only reason for iteration.

The main reason that framework design iterates is because frameworks are supposed to be reusable; all software requires iteration before it becomes reusable. This follows from the general observation that software never has a desirable property unless the software has been carefully examined and tested for it. The ultimate test for whether software is reusable is to reuse it. It is not possible to reuse software until it is written and working, so iteration is inevitable. The only exception is that software that is a reimplementation of existing reusable software might not need iteration. This is because the new software is really a version of the old, and the iteration took place when the old version of the software was designed.

A second reason that framework design iterates is that a framework makes explicit the parts of a design that are likely to change (like `cache` in caches) and the parts that are not likely to change (like the separation of the cache from the data that is in the cache). Features that are likely to change are implemented by abstract operations so that they can be changed by replacing a component of the framework. Interfaces among objects and shared invariants are harder to change. Experience gained by using a framework is one of the most common ways of learning what must be easy to change.

A third reason that framework design iterates is that both frameworks and abstract classes are usually designed by generalizing from concrete examples. A framework is a theory about how to solve problems in a particular application domain; its range of applicability depends heavily on the examples on which it is based. Each example that is considered makes the framework or abstract class more general and reusable. Abstract classes are small, so it is easy to generate lots of examples (*i.e.,* concrete classes) on paper and reduce the chance of iteration. Frameworks are large, so it is too expensive to look at many examples, and paper designs are not sufficiently detailed to evaluate the framework. Thus, a better notation for describing frameworks would probably let more of the iteration take place during the design of the framework.

A common mistake is to start using a framework while its design is still iterating. The more an immature framework is used, the more it changes. This will cause the applications that use it to change, too. It is better to first use the framework for some small pilot projects to make sure that it is sufficiently flexible and general. If it is not, these projects will be good test cases for the framework developers. A framework should not be used widely until it has proven itself, because the more widely a framework is used, the more expensive it is to change it.

The best frameworks are the work of many people. Multiple points of view are needed to learn what is likely to change. Users test the reusability of the framework and provide examples for generalization. The dialog between users and providers of a framework plays an important role in its development. Although the *Choices* virtual memory framework had a single main author, users of the framework helped find many weaknesses and suggested improvements.

This does not mean that frameworks should be designed by committee. A good framework has a conceptual integrity that is usually achieved only by a single person or a small group. However, frameworks arise out of a community of domain experts, and it is not possible for someone who is not closely connected with the application domain to design a good framework.

Because frameworks require iteration and deep understanding of an application domain, it is hard to create them on schedule. Thus, framework design should never be on the critical path of an important project. This suggests that they should be developed by advanced development or research groups, not by product groups.

On the other hand, framework design must be closely associated with the application developers. The purpose of a framework is to make it easier to develop applications. Framework design requires expertise in the application domain. Building applications with a framework shows which parts of the framework need to be improved. Thus, the designers of a framework should collaborate closely with application developers.

This tension between framework design and application design leads to two models of the process of framework design. One model has the framework designers also design applications, but they divide their time into phases when they extend the framework by applying it and a phase when they revise the framework by consolidating earlier extensions [Foo91]. The other model is to have a separate group of framework designers. The framework designers test their framework by using it, but also rely on the main users of the framework for feedback.

The first model ensures that the framework designers understand the problems with their framework, but the second model ensures that framework designers are given enough time to revise the framework. The first model works well for small groups whose management understands the importance of framework design and so can budget enough time for revising the framework. It is essentially the way *Choices* was developed. The second model works well for larger groups or for groups developing a framework for users outside their organization, but requires the framework designer to work hard to communicate with the framework users. This seems to be the model most popular with industry. For example, it was the model used by Apple to develop MacApp.

A compromise is to develop a framework in parallel with developing several applications that use it. Although this will not benefit these first applications much, the framework developers usually help more than they hurt. The benefits usually do not start to show until the third or fourth application, but this approach minimizes the cost of developing a framework while providing the feedback that the framework developer needs.

9 Conclusion

Object-oriented programming is not a panacea. It provides a way to express a design so that it can be instantiated, customized, and extended. However, the hard part of building a reusable design is understanding the problem domain. Object-oriented programming makes it easier to express and communicate a design, but developing new designs always has been and always will be hard.

Object-oriented programming has not been studied much by the software engineering community as a way to reuse design, and there are many topics that deserve attention. These include

- the development of particular frameworks,
- the development of notation for expressing design-level concepts such as abstract classes and frameworks, and
- tools for facilitating the iteration that seems to be an inevitable part of the design of frameworks.

The design of particular frameworks is important both intellectually and commercially. Framework design requires a deep understanding of the problem domain, so a framework contains a great deal of intellectual content. A good framework is valuable, because it can help programmers develop applications quickly and reliably. Thus, we believe that software researchers should put more emphasis on developing frameworks for new application domains and on improving existing frameworks.

Current object-oriented languages do not express design-level concepts like abstract classes and frameworks as well as they should. Statically typed languages can describe the static interfaces between objects, *i.e.,* the set of operations that one can perform on the other, but not the dynamic interfaces, *i.e.,* the order of these operations. One promising approach to describing dynamic interfaces is contracts [HHG90].

Since iteration is so important, framework design would be easier if it were easier to make iterative changes. Most of the changes to frameworks seem to fall into about a dozen categories [JF88, OJ90]. It seems feasible to provide tools to automate these kinds of changes. This would make it easier to change frameworks when weaknesses are discovered, but would not make it any easier to spot weaknesses or to decide how to fix them. These essential activities of design require human creativity and insight and are not likely to be automated in the foreseeable future.

Object-oriented programming is a practical way to express reusable designs. It deserves the attention of both the software engineering research community and practicing software engineers. There are many open research problems associated with better ways to express and develop reusable object-oriented designs, but design techniques such as abstract classes and frameworks have already shown themselves to be useful.

Acknowledgements

Roy Campbell has been the leader of the *Choices* project, and a large number of graduate students have worked on it. We would like to especially thank those who have commented on earlier versions of this paper, including Dave Dykstra and Peter Madany.

Gary Frankel, Brian Foote, Craig Hubley, Mitch Lubars, Carl McConnell, Bill Opdyke, Will Tracz, Rebecca Wirfs-Brock, and Larry Zurawski also gave helpful comments on earlier versions of this paper.

The first author was supported by the Illinois Software Engineering Project, which was funded by AT&T, and by NSF grant CCR-8715752. The second author was supported by NSF grant CISE-1-5-30035 and by NASA grants NSG1471 and NAG 1-163.

References

[AEL88] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 11–20, 1988.

[AL91] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96—107, 1991.

[Bat88] D.S. Batory. Concepts for a database system compiler. *Principles of Database Systems*, 1988.

[BB91] D.S. Batory and J.R. Barnett. *DaTE: The Genesis DBMS Software Layout Editor.* Technical Report, Department of Computer Sciences, University of Texas at Austin, 1991.

[BBR*89] D.S. Batory, J.R. Barnett, J. Roy, B.C. Twichell, and J. Garza. Construction of file management systems from software components. In *Proceedings of COMPSAC 1989*, 1989.

[BDS91] H. Boehm, A. Demers, and S. Shenker. Mostly Parallel Garbage Collection. 1991.

[BR87] Ted J. Biggerstaff and Charles Richter. Reusability framework, assessment, and directions. *IEEE Software*, 4(2), March 1987.

[BS88] Lubomir Bic and Alan C. Shaw. *The Logical Design of Operating Systems.* Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[Che88] David R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):314–333, March 1988.

[CRJ87] Roy H. Campbell, Vincent F. Russo, and Gary M. Johnston. The design of a multiprocessor operating system. In *Proceedings of the USENIX C++ Workshop*, 1987.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[Deu87] L. Peter Deutsch. Levels of reuse in the Smalltalk-80 programming system. In Peter Freeman, editor, *Tutorial: Software Reusability*, IEEE Computer Society Press, 1987.

[Deu89] L. Peter Deutsch. Design reuse and frameworks in the Smalltalk-80 programming system. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability, Vol II*, pages 55–71, ACM Press, 1989.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, Massachusetts, 1990.

[Foo88] Brian Foote. *Designing to Facilitate Change with Object-Oriented Frameworks*. Master's thesis, University of Illinois at Urbana-Champaign, 1988.

[Foo91] Brian Foote. The lifecycle of object-oriented frameworks: a fractal perspective. 1991. University of Illinois at Urbana-Champaign.

[Gol84] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Reading, Massachusetts, 1984.

[Gos90] Sanjiv Gossain. *Object-Oriented Development and Reuse*. PhD thesis, University of Essex, UK, June 1990.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Massachusetts, 1983.

[Hal85] Robert Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[HFC76] A. N. Habermann, L. Flon, and L. Cooprider. Modularization and hierarchy in a family of operating systems. *Communications of the ACM*, 19(5):266–272, May 1976.

[HHG90] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. In *Proceedings of OOPSLA '90*, pages 169–180, October 1990. printed as SIGPLAN Notices, 25(10).

[JC89] Gary M. Johnston and Roy H. Campbell. An object-oriented implementation of distributed virtual memory. In *Proceedings of the Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pages 39–57, Ft. Lauderdale, Florida, October 1989.

[JF88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[JM91] Ralph E. Johnson and Carl McConnell. *The RTL System: A Framework for Code Optimization*. Technical Report, Department of Computer Science, University of Illinois at Urbana-Champaign, 1991.

[Kou91] Panagiotis Kougiouris. *The I/O Subsystem of an Object-Oriented Operating System*. Master's thesis, University of Illinois at Urbana-Champaign, 1991.

[KP88] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August/Spetember 1988.

[KRT89] Shmuel Katz, Charles A. Richter, and Khe-Sing The. Paris: a system for reusing partially interpreted schemas. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability, Vol I*, pages 257–273, ACM Press, 1989.

[Lad89] David A. Ladd. *Supporting Futures With Virtual Memory*. Master's thesis, University of Illinois at Urbana-Champaign, 1989.

[LH86] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, 1986.

[LH87] Mitchell D. Lubars and Mehdi T. Harandi. Knowledge-based software design using design schemas. In *Proc. 9th Intl. Conf. on Software Engineering*, pages 253–262, March 1987.

[LP91] Wilf R. LaLonde and John R. Pugh. *Inside Smalltalk, Volume II*. Prentice Hall, 1991.

[LVC89] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing user interfaces with InterViews. *Computer*, 22(2):8–22, February 1989.

[MCRL89] Peter W. Madany, Roy H. Campbell, Vincent F. Russo, and Douglas E. Leyens. A class hierarchy for building stream-oriented file systems. In Stephen Cook, editor, *Proceedings of the 1989 European Conference on Object-Oriented Programming*, pages 311–328, Cambridge University Press, Nottingham, UK, July 1989.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, 1988.

[MLRC88] Peter Madany, Douglas Leyens, Vincent F. Russo, and Roy H. Campbell. A C++ class hierarchy for building UNIX-like file systems. In *Proceedings of the USENIX 1988 C++ Conference*, pages 65–79, Denver, Colorado, October 1988.

[OJ90] William F. Opdyke and Ralph E. Johnson. Refactoring: an aid in designing application frameworks and evolving object-oriented systems. In James TenEyck, editor, *Proceedings of Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, pages 145–160, September 1990.

[PHK*88] A. J. Palay, W. J. Hansen, M.L. Kazar, M. Sherman, M.G. Wadlow, T.P. Neuendorffer, Z. Stern, M. Bader, and T. Petre. The Andrew Toolkit—an overview. In *USENIX Association Winter Conference*, Dallas, 1988.

[PS85] James L. Peterson and Abraham Silberschatz. *Operating System Concepts.* Addison Wesley, 1985.

[PS88] Inc. ParcPlace Systems. *Smalltalk-80 Reference Manual.* 1988.

[RAN88] M. Rozier, V. Abrossimov, and W Neuhauser. *CHORUS-V3 Kernel Specification and Interface, Draft.* Technical Report CS/TN-87-25.10, CHORUS Systems, February 1988.

[RC89] Vincent Russo and Roy H. Campbell. Virtual memory and backing storage management in multiprocessor operating systems using class hierarchical design. In *Proceedings of OOPSLA '89*, pages 267–278, New Orleans, Louisiana, September 1989.

[RJC88] Vincent Russo, Gary Johnston, and Roy Campbell. Process management and exception handling in multiprocessor operating systems using object-oriented design techniques. In *Proceedings of OOPSLA '88*, pages 248–258, November 1988. printed as SIGPLAN Notices, 23(11).

[RMC90] Vincent F. Russo, Peter W. Madany, and Roy H. Campbell. C++ and operating systems performance: a case study. In *Proceedings* *of the USENIX C++ Conference*, pages 103–114, San Francisco, California, April 1990.

[RTY*87] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine–Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systmes (ASPLOS II)*, pages 31–39, 1987.

[Rus90] Vincent F. Russo. *An Object-Oriented Operating System.* PhD thesis, University of Illinois at Urbana-Champaign, October 1990.

[Sch86] Kurt J. Schmucker. *Object-Oriented Programming for the Macintosh.* Hayden Book Company, 1986.

[Sha87] Robert A. Shaw. *Improving Garbage Collector Performance in Virtual Memory.* Technical Report CSL-TR-87-323, Stanford University, March 1987.

[Sha90] Mary Shaw. Prospects for an engineering discipline of software. *IEEE Software*, 7(6):15–24, November 1990.

[SMC90] Aamod Sane, Ken MacGregor, and Roy Campbell. Distributed virtual memory consistency protocols: design and performance. In *IEEE Workshop on Experimental Distributed Systems*, October 1990.

[Tan87] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation.* Prentice-Hall, 1987.

[VK89] Dennis M. Volpano and Richard B. Kieburtz. The templates approach to software reuse. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability, Vol I*, pages 247–255, ACM Press, 1989.

[VL89] John M. Vlissides and Mark A. Linton. Unidraw: a framework for building domain-specific graphical editors. In *Proceedings of the ACM User Interface Software and Technologies '89 Conference*, pages 81–94, November 1989.

[Vli90] John M. Vlissides. *Generalized Graphical Object Editing.* PhD thesis, Stanford University, June 1990.

[WGM88] A. Weinand, E. Gamma, and R. Marty. ET++: an object-oriented application framework in C++. In *Proceedings of OOPSLA '88*, pages 46–57, November 1988. printed as SIGPLAN Notices, 23(11).

[WGM89] A. Weinand, E. Gamma, and R. Marty. Design and implementation of ET++, a seamless object-oriented application framework. *Structured Programming*, 10(2):63–87, 1989.

[Wir90] Allan Wirfs-Brock. Ecoop/oopsla'90 panel on designing reusable frameworks. October 1990.

[WJ90] Rebecca J. Wirfs-Brock and Ralph E. Johnson. Surveying current research in object-oriented design. *Communications of the ACM*, 33(9):104–124, 1990.

[YT86] Yasuhiko Yokote and Mario Tokoro. The design and implementation of concurrentsmalltalk. In *Proceedings of OOPSLA '86*, pages 331–340, November 1986. printed as SIGPLAN Notices, 21(11).

[ZJ90] Jonathan M. Zweig and Ralph E. Johnson. The conduit: a communication abstraction in C++. In *Proceedings of the Second Usenix C++ Conference*, April 1990.