Pi: A Case Study in Object-Oriented Programming

T. A. Cargill
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Pi is a debugger written in C++. This paper explains how object-oriented programming in C++ has influenced Pi's evolution. The motivation for object-oriented programming was to experiment with a browser-like graphical user interface. The first unforeseen benefit was in the symbol table: lazy construction of an abstract syntax-based tree gave a clean interface to the remainder of Pi, with an efficient and robust implementation. Next, though not in the original design, Pi was easily modified to control multiple processes simultaneously. Finally, Pi was extended to control processes executing across multiple heterogeneous target processors.

1. Introduction

The subject of this paper is the impact of object-oriented programming on the structure and capabilities of a complicated piece of software a debugger called Pi. The paper begins with observations about debugging in general and a description of Pi, to motivate the introduction of object-oriented programming. When the design of Pi began, object-oriented programming was chosen as a means of achieving a style of user interface. But the application of object-oriented techniques throughout had unforeseen benefits with respect to symbol table structure, multi-process debugging and target environment independence.

2. User Interfaces for Debugging

A debugger must provide many views of its subject. The primary view of the subject program is static: the source text. The primary view of the subject process is dynamic: the callstack, a sequence of activation records. The programmer's insight into the program's behavior comes from merging these views: examining data within the process while controlling its progress through the source text. Debugging purely at the source level may be sufficient for a safe language, correctly implemented, but in practice we also need views of the implementation. Even if none of the subject is written in assembler, it is vital from time to time to consider the process in terms of the instruction stream, registers and uninterpreted memory.

A particular view is not used in isolation; it is related to other views. The programmer moves rapidly among a set of related views. This diversity complicates the user interface. Any attempt to base an interface on a coherent model is confounded by the user's need to switch among so many projections of the underlying subject. For example, the keyboard input

```
100
```

could mean ‘‘display the context of line 100’’ of the current source file. It could also mean ‘‘evaluate the expression 100’’ or ‘‘display the value of the memory cell at location 100’’ and so on. There is no natural interpretation; it depends on the programmer's current focus. If there is some simple model on which a user interface for such a (necessarily) powerful tool could be based, no one has yet found it.

Switching among a set of input modes might help as long as there is no confusion over what mode is current and how one changes mode. However, experience with just two modes in text editors suggests that a larger number of modes would not work. On the other hand, modeless keyboard languages for debuggers tend to need many qualifiers and options for each command, varying from verbose to cryptic and from pedantic to treacherous. But if they are useful, they are large and complicated. Two recent debuggers use programmability of the user interface to permit the custom definition of keyboard languages for different classes of users. Kraut [4] (with ‘‘path expressions’’) and Dbx [8] (with macros) let new commands be defined in terms of a base language. In contrast, Pi's user interface uses multi-window bitmap graphics, and cannot be extended by the user.

3. Pi's User Interface

Pi is primarily an attempt to combine expressive power and ease of use in a graphics interface. The user browses through a network of views, each in its own window with a specialized pop-up command menu and keyboard language. The user selects a window by pointing at it with a mouse or by following a menu or keyboard connection from a related window. All the information within a window is textual, i.e., symbolic and numeric rather than analog or geometric. Moreover, each line of text is also part of the browsing network; it defines its own menu and keyboard interface.

The details of the graphics are not essential, but will help make the discussion more concrete. The display has 100 monochrome pixels per inch. Within Pi, windows may overlap and are positioned explicitly by the user. A proportional scroll bar on the left shows how much of a window's text is visible. The current window has a thick border; the current line is video-inverted. A three-button mouse is used. The left button makes selections: to change the current window, to scroll within it or to select a line. The middle and right buttons raise the pop-up menus associated with the current line and window, respectively. A line of accumulated keyboard characters is displayed in a common space below the other windows. If the current line or window accepts keyboard input, its border flashes at each keystroke. At carriage return a complete line of input is sent to the current recipient.

The following example shows much of the user interface mechanics, but only a few of Pi's features. A more complete example appears in [7]. This trivial C program is taken from [12]:

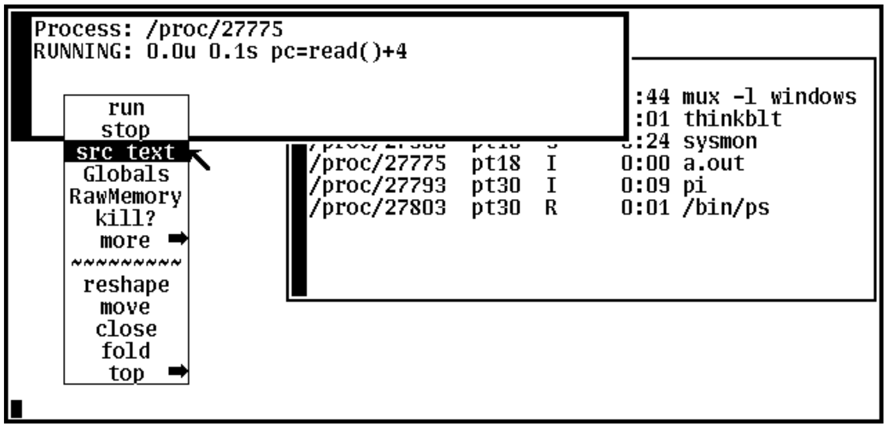
```
#include <stdio.h>

main()     /* count lines in input */
{
    int c, nl;

    nl = 0;
    while ((c = getchar()) != EOF)
        if (c == '\n')
            ++nl;
    printf("%d\n", nl);
}
```

Assume this program has been compiled and invoked by a command interpreter running elsewhere on the screen. The process reads from the keyboard of its virtual terminal. To bind Pi and the process, the user selects the process from a list of accessible processes in Pi's master window. To control this subject process Pi creates a Process window, for which the user must sweep a rectangle with the mouse. The Process window identifies the process and shows its state, as in Figure 1. Process 27775 is in a `read` system call, waiting for keyboard input for the call to `getchar()`.

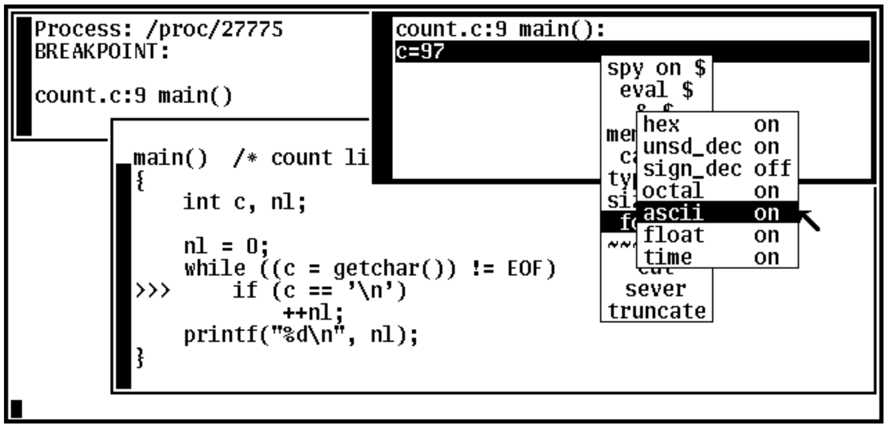
The Process window is the hub of a network of views of the process. Choosing '`src text`' from the Process window's pop-up menu creates a Source Text window for the source file, `count.c`, shown in Figure 2.



```
Process: /proc/27775
RUNNING: 0.0u 0.1s pc=read()+4
```

```
                          :44 mux -l windows
                          :01 thinkblt
                          :24 sysmon
/proc/27775  pt18  I      0:00 a.out
/proc/27793  pt30  I      0:09 pi
/proc/27803  pt30  R      0:01 /bin/ps
```

Figure 1. A Process window and its menu.



```
Process: /proc/27775
BREAKPOINT:

count.c:9 main()
```

```
count.c:9 main():
c=97
```

```
main()  /* count li
{
    int c, nl;

    nl = 0;
    while ((c = getchar()) != EOF)
>>>     if (c == '\n')
            ++nl;
    printf("%d\n", nl);
}
```

Figure 3. A Frame window and an expression's menu at a breakpoint.

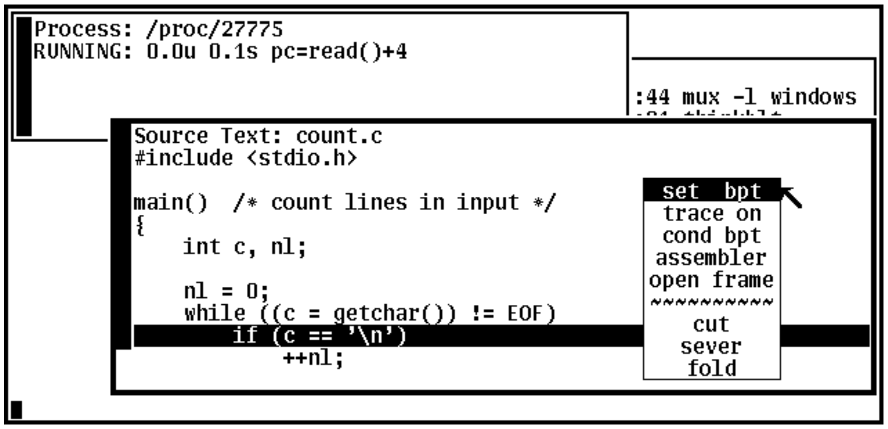
A breakpoint is set by pointing at a source line and choosing '`set bpt`' from the line's menu. The presence of the breakpoint is recorded by >>> at the left of the line (Figure 3).

The subject process reaches the breakpoint when the user supplies some keyboard input and `getchar()` returns a character. The Process window announces the change of state and displays a callstack traceback, in this case a degenerate stack of depth 1. Pi shows the source text context by making the Source Text window current and selecting the source line at which the program stopped. Choosing '`open frame`' from the source line's menu creates a Frame window for evaluating expressions with respect to the activation record associated with that source line.

Expressions in a Frame window are built from menus or the keyboard. Choosing an identifier, say '`c`', from the Frame's menu of local variables creates and evaluates a simple expression, shown in Figure 3. The variable's type determines the default format in which the value is displayed: an integer in decimal. It makes more sense to display `c`'s value as an ASCII character. Changing the format is one of the operations in the expression's menu. Choosing '`format`' from the expression's menu and '`ascii on`' from a sub-menu of formats re-evaluates the expression and displays it as:

```
c='a'=97
```

In debugging a real program there are likely to be many more windows. (Pi is used daily by dozens of programmers, often on programs of 10,000 lines or more.) The windows shown above may be instantiated arbitrarily often: a Source Text window for each of the program's source files and a Frame window for each of the callstack's activation records. There may be only one instance of each of the other window types: a Frame window bound to global scope rather than an activation record, an Assembler window for controlling the process at the instruction



```
Process: /proc/27775
RUNNING: 0.0u 0.1s pc=read()+4
```

```
:44 mux -l windows
```

```
Source Text: count.c
#include <stdio.h>

main()  /* count lines in input */
{
    int c, nl;

    nl = 0;
    while ((c = getchar()) != EOF)
        if (c == '\n')
            ++nl;
```

Figure 2. A Source Text window and a source line's menu.

level, a Raw Memory window for manipulating memory as an uninterpreted array of cells, and a Signals window for controlling process exceptions.

The resemblance between Pi and Dbxtool [1] is superficial, even though both debuggers use windows and menus on a bitmap display. Dbxtool is a front-end that translates graphics commands into the command language of a conventional debugger, Dbx. Dbxtool provides a fixed set of windows, one of which, the ‘‘Command Window,’’ shows the Dbx commands and responses. To see how this affects the user, consider trying to evaluate expressions with respect to two different activations records at once. In Pi, a window is opened for each activation record and the user moves back and forth with the mouse, evaluating expressions in either window; each window then contains a set of related expressions. In Dbxtool, the user enters a mixture of commands that evaluate expressions and move up or down the callstack; the Command Window then contains a transcript of interleaved context changes and evaluated expressions.

Pi is also unlike the debugger in Smalltalk's ‘‘integrated environment,’’ where there is little distinction between the compiler, interpreter, browser and debugger [10]. Smalltalk's tools cooperate through shared data structures and the computer's uniformly defined graphics. On the other hand, Pi is an isolated tool in a ‘‘toolkit environment.’’ Pi interacts with graphics, external data and other processes through explicit interfaces. Pi adapts the graphics techniques exemplified by Smalltalk to the toolkit setting.

4. Object-Oriented Programming

The remainder of the paper concerns the implementation of Pi. The object-oriented programming model is better suited to the implementation of Pi than the sequential programming model or the multiple sequential processes model.

Under the sequential model, a process executes a program. At any time the process is in some state, with control at some location in the program. When the process blocks for input from the user it is in the state from which it resumes when the user responds. This makes it difficult to drive the process with the flexibility described above, where the user is allowed to refuse a given menu and leap to some unrelated context. The process must accept either the menu selection or the context switch. The menu selection is a local operation in the local context; the context switch is a global operation involving unrelated parts of the program. This might be achieved by letting the user traverse a network of contexts. But there is then a tradeoff between ease of use and ease of programming: a tree is easy to program, but tedious for the user to traverse; a fully connected network might be easy to use, but calls for every part of the program to be intimately coupled to every other part.

With multiple sequential processes a separate process could be associated with each context. As the user moves around the graphics screen, input is directed to the appropriate process by some agent that maps screen locations to processes. Processes are suspended until the user needs them; they need know only about semantically related processes. As a model this yields a natural architecture, but its implementation is usually very expensive. The overhead for creation and interaction of separate processes might be acceptable for a small number of processes, say one per window. But the desired user interface has each line within each window interacting independently with the user. Many hundreds, even thousands, of processes would be needed. With conventional multi-process techniques the cost is prohibitive.

The object-oriented model lies between these two models. Instead of a collection of processes there is a single process containing a collection of *objects*, each an instance of a type called a *class*. Each object has a copy of the *data members* defined in its class and can execute the *function members* of the class. Unlike a process, an object has no permanent state other than its data. Objects share a universal address space and communicate with one another by invoking function members as procedures. It is feasible to have many thousands of objects. That objects cannot execute concurrently, as processes do by timeslicing a physical processor, is not significant; such concurrency is not required.

The C++ [14] programming language supports this model. C++ is a superset of C with Simula-like classes [3]. The members of a class are data and functions, in private and public sections:

```
class identifier {
        private_data_declarations
        private_function_declarations
public:
        public_data_declarations
        public_function_declarations
};
```

(This grammar generates only a subset of C++ class declarations.) The example of C++ code below defines a class `Const` that privately represents an integer value; the value is rendered as a hexadecimal, decimal or octal character string by three member functions. Similar code is found in Pi.

```
class Const {
        int      val;              // private data member
public:
                 Const(int);       // constructor
        char     *hex();           // public function members
        char     *dec();
        char     *oct();
};

void Const::Const(int v)           // constructor body
{
        val = v;
}

char *Const::hex()                 // function member body
{
        build character string
        return pointer to character string;
}

 ...
```

A member function whose name is the same as that of its class is a *constructor*. If defined, the constructor is invoked to initialize a new object. `Const`'s constructor assigns its argument to the private representation of the value. `Const`'s member functions `hex()`, `dec()` and `oct()` return pointers to ASCII representations of the value. The following client code prints 123 as `0x7B=123=0173`:

```
{
        Const c(123);             // c.Const(123) called implicitly
        printf("%s=%s=%s", c.hex(), c.dec(), c.oct());
}
```

Here `Const c(123)` declares an object instantiated on the stack for the lifetime of the block; the argument `123` is passed to the constructor. The object could also be allocated from the free store by means of the operator `new`, in which case the variable's type is pointer-to-`Const`:

```
{
        Const *cp;
        cp = new Const(123);       // allocate from free store
        printf("%s=%s=%s", cp->hex(), cp->dec(), cp->oct());
        delete cp;                 // return to storage pool
}
```

In general, an executing C++ program consists of a network of objects, referencing one another with pointers. Subject to the encapsulation rules of C++, objects may access one another's member data and invoke member functions. Consider a Frame window that evaluates expressions with respect to an activation record. Figure 4 shows two objects: `F`, an instance of class `Frame`, and `E`, an instance of class `Expr`. An arrow represents a pointer; an arrow labeled by a function call indicates that the holder of the pointer may call the member function of the object to which it points.

Here, `create_expr()` is a member of `Frame` and `evaluate()` is a member of `Expr`:

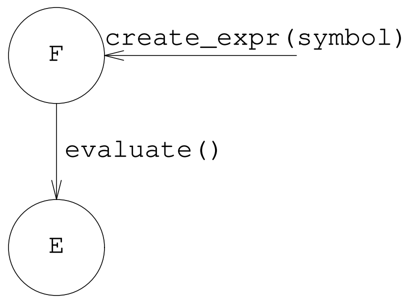


Figure 4. A sub-network of objects.

```
class Frame{
        ...
        void create_expr(Symbol*);
        ...
};

class Expr{
        ...
        ErrMsg evaluate();
        ...
};
```

The `Symbol*` argument to `create_expr` is a pointer to a symbol table object describing a variable in `F`'s activation record's scope, from which the `Frame` creates an `Expr` object `E`. `F` might cause the expression to evaluate itself by calling `E.evaluate()`, and so forth.

The notation in Figure 4 suggests a message-passing model. Indeed, the terminology of object-oriented programming in Smalltalk [10] refers to ''messages'' between objects. However, the communication is by procedure call.

5. Object-Oriented User Interface

A user interface to such a network can be created by letting the user communicate directly through the interfaces that objects present to one another [10]. The user must be able to select an object, see a set of member function calls and select one to be invoked. To receive a member function call from the user an object must describe itself and a set of function calls to the software managing the display.

Figure 5 shows the object network of Figure 4 and a screen image. `F` has associated itself with a window on the screen, and `E` has associated itself with a line in that window. The new ''pointers'' from images on the screen to objects in the program are shown by dashed arrows. If the user selects `E`'s line, raises the line's menu and selects '`ascii on`', then the result is a call of `E`'s member function:

```
E.reformat(ASCII_ON)
```

Three points of detail should be mentioned. First, the user does not really see the program's internal interfaces, i.e., the identifiers and types of member functions. The text of a menu entry is chosen to support the user's view of the function and need not reflect program's source literally. For example, the user sees '`ascii on`' instead of '`reformat(ASCII_ON)`'. Second, when invoked, a member function cannot distinguish a call originated directly by the user from a call by another object. There is only one procedure call mechanism. Third, once an object has exposed its image and a set of member function calls to the user, it must be prepared to receive any sequence of calls the user chooses to make. To inhibit the user, the object must remove itself or change the set of calls in the menu. Changes in the menu reassure the user about the state of the object: the menu of a line of source text on which the user has already set a breakpoint shows '`clear bpt`' instead of '`set bpt`'.

The interaction between the user and program is now conducted without a ''user interface'' in the conventional sense. There is no part of the program responsible for reading, parsing and interpreting a sequence of commands from the user. The user sees a graphical image of the program and the program sees the user as an active participant in its object-oriented world, with implicit software mapping between the two.

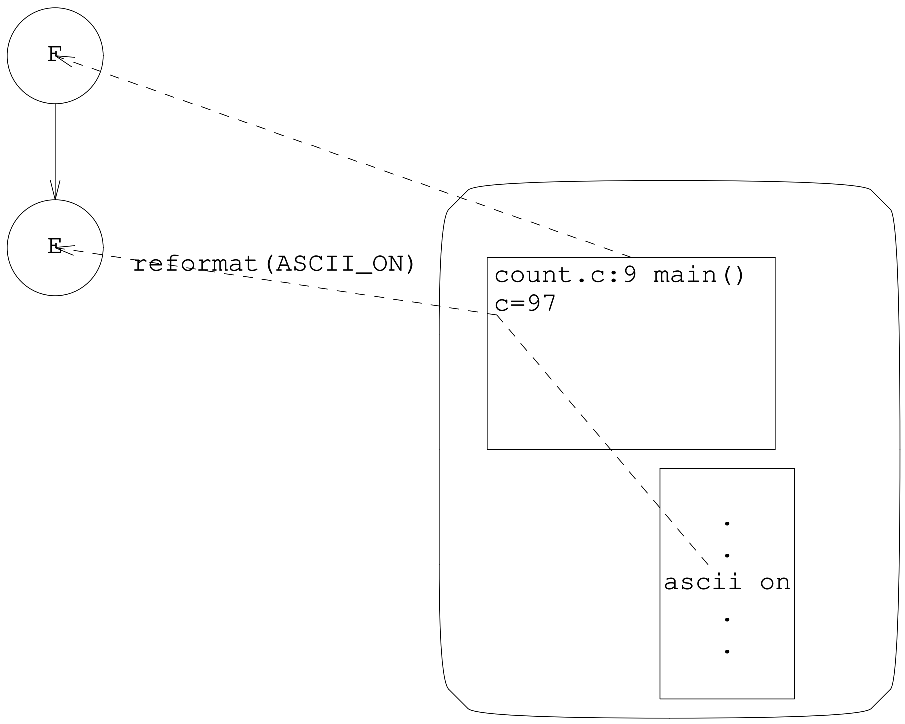


Figure 5. Operations invoked by the user.

6. Implementation

The interface between the graphics and the network of objects is a pair of C++ classes: `Window` and `Menu`. These classes in turn interact with a graphics package. To create its window, the `Frame` object executes:

```
w = new Window;             // Save the pointer to a new
                            // Window in Frame’s data member w.
w->bind(this);              // Pass a pointer to this (a C++ keyword)
                            // instance of Frame to the Window.
w->title(description());    // Frame’s member function description()
                            // yields a string like "count:9 main()".
w->makecurrent();           // To make a newly created window current
                            // the user must sweep a rectangle for it.
```

To display its value in the window an `Expr` object executes:

```
Menu m;                     // Instantiate a Menu on the local stack.
 .
 .
m.append("octal on", &reformat, OCTAL_ON);
m.append("ascii on", &reformat, ASCII_ON);       // Build the menu.
m.append("float on", &reformat, FLOAT_ON);
 .
 .
w->insert(line, this, m, evaltext());   // Insert line in window.
```

Each call to `m.append()` adds an entry to the menu. Each entry consists of the text string the user sees when the menu is raised, the member function to be called and the argument to be passed. The call to `insert()`, using a copy of `w` passed to the `Expr` by the `Frame`, creates a new line of text in the window. The text string, like `"c=97"`, is obtained from `Expr`’s member function `evaltext()`. The `line` argument is a numeric key that determines where this line will appear relative to other lines in the window. Not shown here is how ‘`ascii on`’ would be made to appear in a sub-menu, as in Figure 3. Instead of being bound to a window or line of text, one

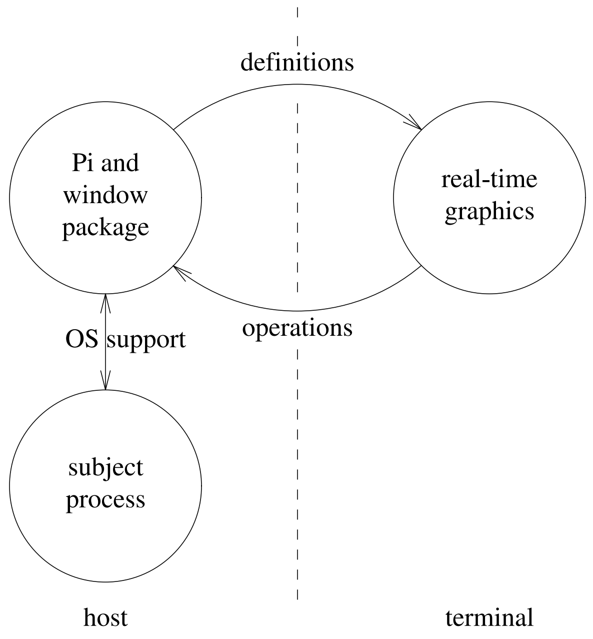


Figure 6. Inter-process communication.

`Menu` may be embedded in another. The embedded sub-menu pops up when the cursor is positioned over a sub-menu indicator in the super-menu.

Note that the code from `Frame` and `Expr` does not reflect the terminal's physical attributes. The abstractions are windows and menus, not coordinates and mouse buttons. Pi has no knowledge of, or control over, details such as the placement of windows, the scrolling of text or the assignment of mouse buttons. Though not used here, the most concrete graphics request is that a line of text be made visible to the user. This means that the graphics and mouse protocol described above is just one of many possible choices; the user's view might be quite different in an implementation for a one-button mouse.

Operations on `Window` objects are translated into a stream of messages passed to a separate process responsible for the real-time graphics. Operations from the user are transmitted back to the main process, received by the package and invoked on the Pi's objects. The two processes execute asynchronously on separate processors: the main process on a UNIX® system, ''the host,'' and the real-time graphics process on a Teletype DMD 5620 bitmap terminal, ''the terminal'' (Figure 6).

Global control in the graphics program lies in a loop that polls the host and user for commands. The following pseudo-code approximates the loop:

```
for(;;){                          // forever
    if( mouse activity ){
        update screen
        if( remote operation selected )
            send message to host
    }
    if( message from host ){
        receive from host
        update screen
    }
    if( real-time clock event ){   // see below
        send message to host
    }
}
```

Global control in the host process lies in a loop in the window package that blocks waiting for messages from the terminal:

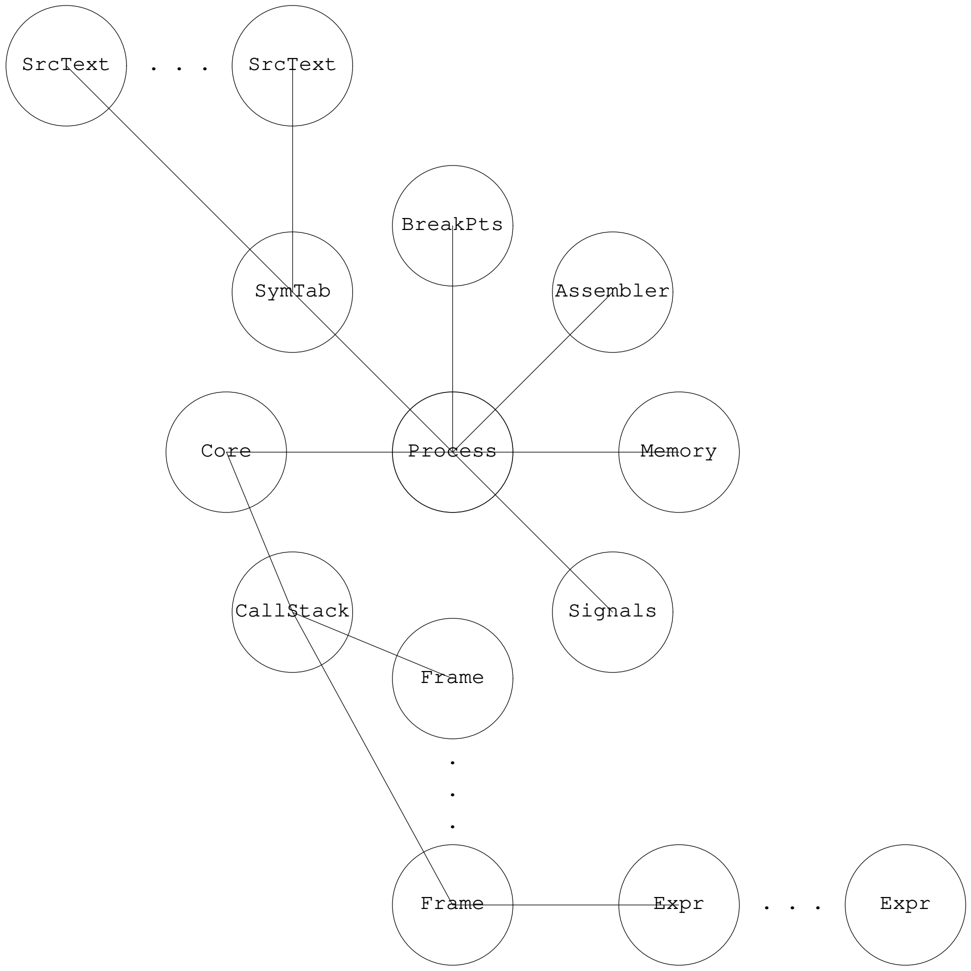


Figure 7. Pi's object network

```
for(;;){
        read <object, operation, operand> from terminal
        invoke object->operation(operand)
}
```

Usually, an invoked operation in turn causes definitions or redefinitions of objects to be sent to the terminal, to show the user some kind of result. The messages from the host arrive asynchronously with respect to the user's interaction with the graphics. An experienced user need not wait for changes from each operation to be reflected on the screen before invoking another operation. For example, having requested a particular context within a source file to be displayed, the user can select a source line and set a breakpoint on it as soon as that line is visible, even if the host has not finished sending all of the lines needed to fill the window. Some users take advantage of the asynchrony as the natural way to function; others operate as though the communications were half-duplex.

7. Pi's Architecture

The debugger is a network of objects as described above. An object of class `Process` is created to take overall control of the subject process. The `Process` object creates two major objects to serve it: a symbol table object of class `SymTab` and a core image access object of class `Core`. The `SymTab` is responsible for reading the symbol table as left by the compiler, assembler and loader and providing that information to the rest of the debugger, as discussed below. The `Core` is responsible for all access to the address space of the subject process and the operating system's control information. Details of the physical processor, operating system and compiler generated code are encapsulated in `Core`.

Of these three classes, only `Process` opens a window, the Process window, from which the user may then open other windows. To display a callstack, the `Process` obtains a `CallStack` object from the `Core` and extracts each activation record it needs as a `Frame` object from the `CallStack.` If a `Frame` is referenced by the user, it opens a Frame window. As the user creates expressions and derives new ones, each expression is an `Expr` object which displays itself as a line in its `Frame`'s window. Figure 7 shows the network. The lines indicate the primary permanent links between objects, but pointers are passed around as needed.

It is the `Process` that monitors the state of subject process. When the subject is running, its state must be polled to see if it reaches a breakpoint or some other exception. The `Process` therefore periodically executes an operation that reads the current state of the subject from the `Core`. This operation re-invokes itself by sending to the terminal a message requesting that the terminal invoke it after a specified delay. The invocation returned from the terminal is interleaved with, and indistinguishable from, invocations made directly by the user. The `Process` can monitor the state of the subject process while the user asynchronously evaluates expressions, sets breakpoints and so on. This technique creates a few bytes per second of extra host-terminal traffic, since it would be possible to use a clock interrupt on the host instead. However, to guarantee that operations are invoked fairly is much simpler if the only source of operations is a single stream coming from the terminal. Also, the terminal is a better place for real-time programming, both in terms of operating system support and expendable processor resources.

8. Symbol Tables

A debugger's needs of its symbol table are similar to those of a compiler, for example, to determine the variables in scope at some point in the program. If the symbol tables prepared by the translator suite reflect the data structures used in the compiler, the debugger is much simplified [5]. If the tables have been ‘‘flattened’’ by an assembler or loader, the debugger suffers the loss of information. Reconstructing an acceptable data structure can be very time-consuming [2]. Working with the flattened tables complicates those parts of the debugger that make non-trivial use the symbol table [6].

For Pi, it seemed likely that versions of the debugger would be used with compilers and assemblers that produced at least two distinct flattened formats. So the first goal was to find a format-independent internal representation that could be built from either format and was well-suited to debugging. Given that the debugger was being written in C++, it seemed a good opportunity to experiment with object-oriented programming. In contrast to the user interface, there was no overall design paradigm guiding the symbol table; the software evolved as different ideas were tried.

From the outset, the symbol table was a sub-network of objects. Its structure follows the abstract structure of the subject program a tree. The root of the tree is the `SymTab` object. It reads the file prepared by the loader and builds a tree, as shown in Figure 8. Below the `SymTab` there is a `SrcFile` object for each separately compiled source file that contributed to the program. (There is a one-to-one correspondence between the `SrcFiles` and the `SrcTexts` of Figure 7; a `SrcText` is created and opens a window when the user decides to examine the source text from the corresponding `SrcFile`.) Below each `SrcFile` there is a `Function` for each function defined in that file. Below each `Function` there is a `Block` of arguments and a `Block` of local variables. Each `Block` has a list of `Variables`. Each `Function` also has a list of `Statements`, the source statements in the function. Some symbols can also be accessed directly from a hash table whose entries point into tree's interior.

Like any object, a member of the symbol table can receive operations from the user. For example, the object associated with a line of source text in a Source Window is a `Statement`. To set a breakpoint the user communicates directly with the symbol table.

Though not shown in the figure, each node in the tree has four pointers: to its parent, leftmost child and right and left sibling. These make it easy to traverse the tree. C++'s type inheritance is used to capture the common properties of all symbol nodes in a *base class* called `Symbol` from which other classes of symbols are *derived*:

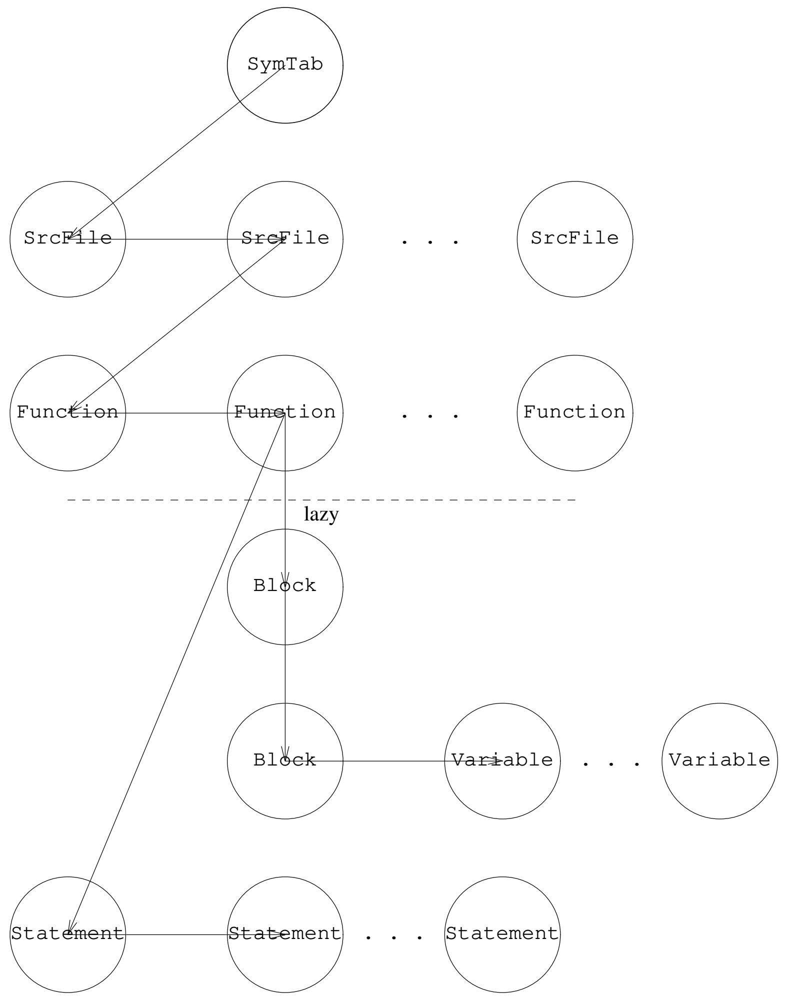


Figure 8. The symbol table hierarchy

```
class Symbol {
public:
        Symbol  *parent;
        Symbol  *leftmost_child;
        Symbol  *right_sibling;
        Symbol  *left_sibling;
        Address addr;
        char    *id;
virtual char    *text();
};
```

The `addr` and `id` data members record address information and an identifier for each symbol. The function `text()` returns a textual representation of the node, which is just the identifier:

```
char *Symbol::text()
{
        return id;
}
```

There are no instances of class `Symbol` in the tree; each node of the symbol table is of a type derived from `Symbol`. For example:

```
class Variable : public Symbol {
public:
        Storage     storage;
        DataType    type;
};
```

As a derived class, `Variable` inherits all the data and functions of `Symbol`; it has two additional data members specific to its needs. In `Symbol`, the function `text()` is declared *virtual.* This means that a derived class may override the base version with its own. `Variable` has no need to do this; it is adequately served by the base version that returns the identifier. However, `Statement` does define its own `text()`:

```
class Statement : public Symbol {
        ...
        int     line_number;
        char    *text();
};
```

Instead of returning the identifier (which is not used by `Statement`), `Statement::text()` walks up the tree to find its `SrcFile` node and returns a string identifying the statement's source file and line number, like:

```
"count.c:9"
```

In general, code traversing the tree to extract textual information does not depend on whether the class of a given node defines its own version of `text()`. For example, `Statement::text()` applies `text()` to a `SrcFile` pointer to obtain the pathname of the source file to embed in the string being built; it does not know whether `SrcFile` has defined its own version of `text()`. The choice of implementation of an operation by the object on which the operation is invoked, rather than the invoker, is common to all object-oriented programming. The technique is used throughout Pi.

9. Lazy Symbol Table Construction

If the symbol table as described above were really built it would consume enormous time and space. Early versions of Pi did build it and could only be applied to small programs. Time and space performance was improved dramatically by delaying construction of the subtree of local symbols below each `Function` node until needed ‘‘lazy’’ construction.

When the debugger picks up a process it scans the flattened symbol table to build the tree only down to the `Function` level the part of the tree above the dashed line in Figure 8. From a `Function` the only public access to its sub-tree is through function members. The members of `Function` detect that the sub-tree is missing and call on the `SymTab` to build it before returning a pointer to a requested `Block` or `Statement`. Once the sub-tree has been built it remains; pointers into it may have been passed to, and retained by, other objects.

Lazy construction allows large symbol tables to be presented to the rest of the debugger in a manner that is encapsulated naturally and efficiently. The real-time initialization delay is about one second per thousand lines of source text, on a VAX-11/750 processor. The cost of building a sub-tree on demand is negligible and very few are built. Local tables are needed for only those functions on the callstack or visible in a Source window. It is exceptional to need tables for more than about 15 functions. When Pi, a 500-function program, is used to debug itself, it typically builds 1% 5% of the local tables.

This style of lazy table construction could be implemented in any programming language, but reliably only in a language that enforces its data abstraction. In C++, because the only public access from a `Function` to its sub-tree is through member functions, the lazy operation of the symbol table does not depend on cooperation from clients. Very few problems have arisen with this code.

A similar lazy method is used to defer the construction of the tables of user-defined types; the savings are comparable.

10. Generators that Traverse the Symbol Table

Clients of the symbol table need to perform various traversals to extract information. For example, a menu built by a `Frame` contains the local variables visible from a function. This might be implemented by a natural

VAX is a trademark of Digital Equipment.

traversal of the data structure in Figure 8. However, the symbol table can be better encapsulated by providing a class `VisibleVars`, an instance of which performs such a traversal:

```
class VisibleVars {
 ...
public:
                  VisibleVars(Block*);
        Variable  *gen();
};
```

The constructor takes an argument pointing to a `Block`. The only public function, `gen()`, returns a pointer to a different `Variable` on each call, ending with a null pointer. Client code with a pointer to a `Block` from somewhere:

```
Block *b;
```

creates an instance of `VisibleVars` and iterates through the generated variables:

```
{
        VisibleVars vv(b);
        Variable *var;

        while( var = vv.gen() ){
                ...
        }
}
```

This iteration is built on general purpose data abstraction in C++, rather than a built-in iteration primitive [11], [13].

Parts of Pi use nested iteration through the variables visible from a function. Nested iteration arises in the code that warns the user of ambiguity when an identifier occurs more than once in a menu. To determine if an identifier is non-unique, the menu builder searches the identifiers of the variables that its generator will produce later in the iteration by taking a *copy* of the generator in its current state and iterating through the copy:

```
{
        VisibleVars vv(b);
        Variable *v;

        while( v = vv.gen() ){
                ...
                VisibleVars copy(0);  // initialized with null block
                copy = vv;
                Variable *w;
                while( w = copy.gen() ){
                       if( strcmp(v->id, w->id) )
                               ...
                }
                ...
        }
}
```

(In this case quadratic running time is acceptable.)

11. Debugging Multiple Processes

The initial design of Pi did not consider letting the user examine more than a single process at a time. The `Process` class (Figure 7) had been introduced in order to follow the object-oriented paradigm uniformly throughout the program. The intention was to instantiate `Process` only once. Other parts of the debugger were made a little more complicated by this decision because they had to fetch data from the `Process` that might otherwise have been stored in global variables. This version of the debugger did not have a master window or dynamic binding to processes; the subject process was fixed by a command line argument.

Once Pi was working reliably, I realized that with trivial modification it could instantiate an arbitrary set of `Process` objects to examine an arbitrary set of subject processes. The user would need only one copy of the debugger, no matter how many processes were to be examined. It took only a few days to implement. A `Master` object creates a `Process` object for each subject process that the user chooses to examine, as shown in Figure 9. Each `Process` and its sub-network knows nothing of any others that might exist. The user interface package sees no qualitative change just more objects. The technique whereby a `Process` polls its subject is also unaffected; the delayed invocations of the polling operation from each `Process` are interleaved with one another. None of the problems described in [1] of implementing and using a multi-process debugger have been encountered.

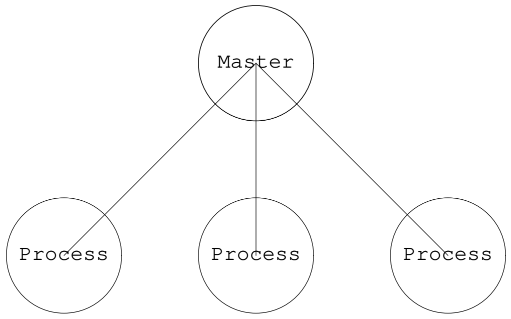


Figure 9. Object network for multi-process debugging.

Multiple process debugging could also have been achieved by instantiating multiple debugger processes rather than multiple `Process` objects within a single debugger. The advantage of instantiating only a single debugger is the reduced overhead for both the user and the computers. With only one instance of the debugger the user has less to manage on the screen. When multiple processes are debugged the set of debugging windows are the same whether they are together in one debugger's window or in separate instances of the debugger. But when the user wants to treat the debugging environment as a whole, it is better to deal with a single tool. For example, removing a single debugger is simpler than removing a set of debuggers. Less machine resources on the host and terminal are required to execute only a single process in each. In special circumstances (such as debugging a debugger), multiple instances of the debugger are needed.

12. Multiple Target Environments

The initial design did anticipate versions of Pi that would operate in different target environments. Those parts of the debugger dependent on the target processor were encapsulated in the `Core` and `Assembler` classes, those dependent on the operating system in `Core`, and those dependent on the external symbol table format in `SymTab`. The original version of Pi was for a VAX processor running the Eighth Edition of the UNIX system. The intention was to tailor versions to different target environments as the need arose. The first demand was for a version to debug processes in the DMD 5620 bitmap terminal: an AT&T WE32000 processor running a virtual terminal multiplexor called Mux.

As a further experiment with object-oriented programming, I decided to build both of these versions as a single program so that one instance of Pi could simultaneously examine processes in both target environments. For each target-dependent class there must be a base class, with a derived class for each target environment. Everywhere a target-dependent object is instantiated it must be of the appropriate derived class, but its target-independent clients need not know from which derivation. The classes derived from `Core` are `HostCore` and `TermCore`, for the host and terminal, respectively. The class hierarchy for `Core` is shown in Figure 10.

The main process of Pi is still a process in the timesharing host. To access memory in the terminal, an instance of `TermCore` (executing on the host) communicates with an additional agent process in the terminal. (The agent process need not be in the same terminal as the real-time graphics process, but usually it is.) This communica-

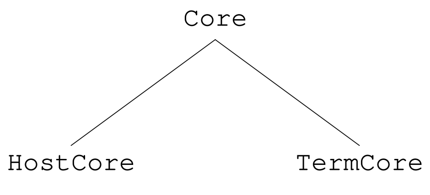


Figure 10. Class hierarchy for `Core`.

tion is based on remote procedure calls from the host to the terminal. The debugger is now three processes altogether: the host process, the real-time graphics process and the terminal access agent, as shown in Figure 11. Implementation experience with a previous debugger [6] indicated that bandwidth between the host and terminal would limit performance. This influenced the design of the host/terminal protocol and satisfactory performance was achieved. For example, when a `TermCore` requests a callstack traceback from the terminal, the terminal computes the callstack locally, compares it to the last callstack sent to the host and transmits the difference usually only the deepest activation record has changed.

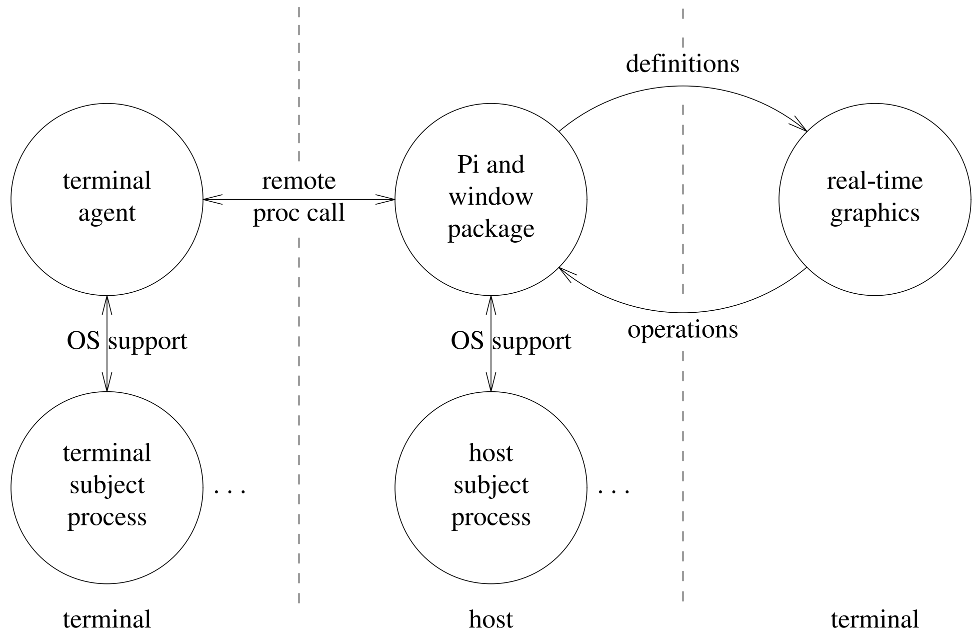


Figure 11. Pi's three processes with two subject processes.

How `Core` finds the value of the subject's program counter is a simple example of inheritance and virtual functions at work. Consider part of the definition of `Core`:

```
class Core {
 ...
public:
virtual int  pc_index();        // register number for program counter
virtual long reg_save(int r);   // address at which register r saved
virtual long peek_long(long a); // fetch value from memory at address a
virtual long pc();              // fetch value of program counter
 ...
};
```

The code here has been somewhat simplified to eliminate irrelevant complications; for example, it doesn't handle errors. `Core::pc()` is target-independent, though it calls the target-dependent functions `pc_index()`, `reg_save()`, and `peek_long()` to obtain its result:

```
long Core::pc()
{
        return peek_long( reg_save( pc_index() ) );
}
```

`pc_index()`, `reg_save()` and `peek_long()` must be implemented for each of `HostCore` and `TermCore`. For example, the program counter is register 15 on the VAX:

```
int HostCore::pc_index()
{
        return 15;
}
```

`reg_save()` and `peek_long()` have the target-dependent code to find the location at which an arbitrary register has been saved and read the contents of an arbitrary memory location, respectively.

Note that `Core::pc()` is virtual; the derived classes *may* also redefine it. So, even though `TermCore` could inherit this functionally correct `pc()` from `Core`, it has its own version. The base version reads memory every time it needs the value of the program counter. For the terminal, this would mean a remote procedure call to the terminal every time. As an optimization, `TermCore` keeps a copy of the program counter, updating it each time the state of the subject process is checked; `TermCore::pc()` simply returns this cached value. The semantics are slightly different: if the program counter is manually patched while the program is halted, `TermCore::pc()` will report the old value. In practice this discrepancy is less significant than the minor differences that arise from operating system idiosyncrasies. No user has ever noticed it.

Finding suitable target-independent base abstractions and implementing the derived classes took several months; simply building a new version of Pi specifically for the new target environment would have taken a few weeks. The `SymTab` class was relatively straightforward, but tedious because of arbitrary differences in the detailed representation of the symbol tables. Two hard parts of finding an acceptable inheritance for `Core` were byte ordering and function calling. The problems encountered with byte ordering are instructive the original scheme did not work on either machine, for reasons that no amount of forethought (by me) would have revealed. The scheme is to read memory from the subject process and create objects from which various types of data (byte, short, long, float, double) can be extracted later by clients of `Core`. The VAX version failed when it tried to set up arbitrary bit patterns as candidates for extraction as floating point values; the operand of a floating move instruction must be a valid floating point number, of which 1 in 256 bit patterns is not. The WE32000 version did not work because the processor does not use the same byte ordering for code and data fetches; a multi-byte constant embedded in code is not the same bit pattern as that constant in data. Neither of these problems was hard to fix, but they indicate the difficulty of finding machine-independent abstractions for hardware. Harder was the interface through which the expression evaluator calls a function in the subject. The mechanisms for the host and terminal are quite different. For the host architecture, the debugger arranges that the subject process execute the function using the user's stack; in the terminal the function is executed directly by the debugger's agent process in the terminal on its own stack. The operation is broken down into a series of steps, each performed by a member of the respective derived `Core`, such that `Expr` can detect no difference. The five steps supplied by `Core` are: save context, allocate argument area, call function, determine location of returned result, restore context.

A further derivation from `HostCore` has been added for examining core dumps from the UNIX kernel on the VAX. `KernelCore` differs very little from `HostCore`; the major change is that memory fetches must be mapped through the kernel's page tables. A derivation of `Core` for S-Net, a multi-processor computer based on the Motorola MC68000, is being implemented at the time of writing. The current `Core` hierarchy is shown in Figure 12.

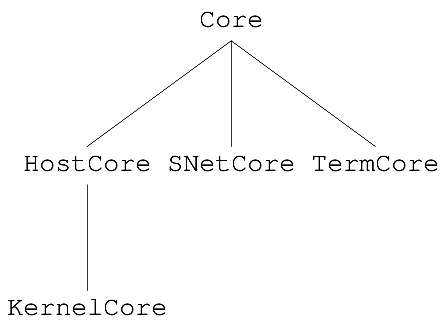


Figure 12. Current `Core` hierarchy.

A single debugger that handles multiple target environments has been a success for both the users and the implementer. The user is guaranteed to see the same interface when debugging in all environments. When changes are made to target-independent parts of Pi they are usually only tested for one target before being installed they almost always work correctly for the others. This was true of adding a trace history of breakpoints, for example. More complicated changes, involving target-dependent parts, take some time before a clean compilation can be achieved, because several derived classes must be changed consistently. It often takes days to get an error-free com-

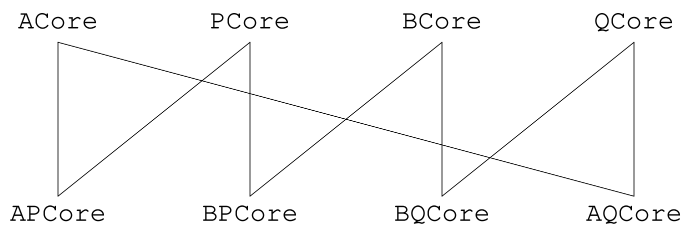


Figure 14. Multiple Inheritance.

pilation. It is frustrating to be unable to test new code for machine X because the code for machine Y is out of date and cannot compile. The discipline introduced is that thought must be given to all target environments simultaneously; this results in earlier exposure of target environment inconsistencies.

13. Deficiencies in C++

Though indispensable in the construction of Pi, C++ is deficient in two respects. First, derived classes may inherit from only a single base class. Second, the benefits provided by classes come at the expense of considerable compilation overhead.

Though each class derived from `Core` is a single step from its parent in the type hierarchy, the step embodies several independent changes: the processor, the operating system and the compiler. If Pi had to support all four target environments possible with operating systems A and B on processors P and Q, the class hierarchy would be that of Figure 13. As a result, each derived class would contain target-dependent code replicated in two others.

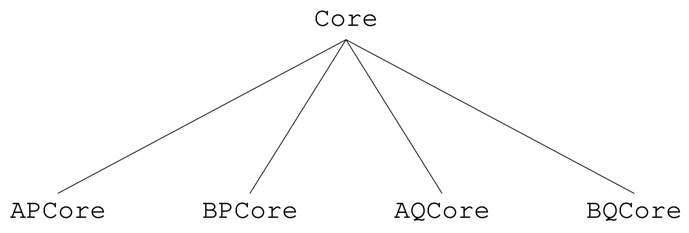


Figure 13. Multiplicity of Derived Classes.

This is a situation in which *multiple inheritance* could be used to eliminate replication. Under multiple inheritance a derived class may inherit from more than one base class. The derivation graph could be as shown in Figure 14. Target-dependent code would be confined to a single appearance in one of the base classes: `ACore`, `BCore`, `PCore` and `QCore`. The derived classes would need no additional code.

No pair of target environments have yet shared a common component, but time will certainly change that. Since C++ does not provide multiple inheritance, some other means of factoring the program must be found to avoid duplicating code. Of course, the success of multiple inheritance cannot be guaranteed without practical experience, but it is certainly worth pursuing.

A more severe problem that has been encountered is the cost of recompilation triggered by the modification of class declarations. The declaration of a class, `X`, places both the public and private components of its interface in a single syntactic unit. The declaration is usually stored in a ''header file,'' `X.h`. Two compilation problems arise with respect to *clients* of `X`, that is, other classes that depend only on `X`'s public interface. Before processing the source text of a client the compiler must read the header file, `X.h`. It therefore reads both the public and private declarations in `X`, even though the client's source is denied reference to the private declarations. (At the implementation level, the client might depend on this private information: to generate client code, the compiler needs to know the *size* of the private data in `X`, if an instance of `X` appears in a client's stack frame.) If the private declarations in `X.h` in turn depend on other header files, they must also be included, and so on. Compilation of the client therefore depends on many header files, even though the client does not need the information from those header files. This makes client compilation more expensive, because of the additional header files that must be processed. Moreover, using conventional Make [9] dependencies, client source is frequently recompiled after the modification of private declarations in classes unreferenced by the client. The subterfuge that partially overcomes this problem is unworthy of description.

14. Conclusion

Object-oriented programming in C++ has worked very well in Pi. Pi's ability to examine multiple processes over multiple target environments follows from the object-oriented model and class inheritance mechanism used in the implementation. At the outset the goal was to experiment with the user interface. Had an object-oriented programming language not been available, I doubt that Pi would have evolved beyond experiments at that level.

15. Acknowledgements

The success of Pi owes much to the ideas and software of Bjarne Stroustrup, Tom Killian and Rob Pike. Thanks also to Brian Kernighan, Doug McIlroy and John Linderman for their comments on drafts of this paper.

16. References

1. Adams, E. and Muchnick, S.S. Dbxtool A Window-Based Symbolic Debugger for Sun Workstations. In *USENIX Summer Conference Proceedings*, USENIX Association, Portland, June, 1985, pp. 213-227.
2. Beander, B. VAX DEBUG: an Interactive, Symbolic, Multilingual Debugger; Proceedings of Symposium on High Level Debugging (Asilomar). In *Sigplan Notices*, Vol. 18, No. 8, 1983, pp. 173-179.
3. Birtwistle, G., Dahl, O., Myrhaug, B., and Nygaard, K. *Simula Begin*. Chartwell-Brat, 1980.
4. Bruegge, B. Adaptability and Portability of Symbolic Debuggers. PhD Thesis, Carnegie-Mellon University, 1985.
5. Cardell, J. Multilingual Debugging with the SWAT High-level Debugger Proceedings of Symposium on High Level Debugging (Asilomar). In *Sigplan Notices*, Vol. 18, No. 8, 1983, pp. 180-189.
6. Cargill, T. Implementation of the Blit Debugger. *Software Practice & Experience 15* (1985), 153-168.
7. Cargill, T.A. The Feel of Pi. In *USENIX Winter Conference Proceedings*, USENIX Association, Denver, January, 1986, pp. 62-71.
8. Computer Science Division, University of California. *Unix Programmer's Manual, Virtual VAX-11 Version*. Berkeley, June, 1981.
9. Feldman, S.I. Make a program for maintaining computer programs. *Software Practice & Experience 9*, 4 (1979), 255-266.
10. Goldberg, A. *Smalltalk-80 The Interactive Programming Environment*. Addison-Wesley, 1984.
11. Griswold, R.E. and Griswold, M.T. *The Icon Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1983.
12. Kernighan, B.W. and Ritchie, D.M. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 1988. 2nd Edition.
13. Shaw, M., Wulf, W., and London, R. Abstraction and Verification in Alphard: Iteration and Generators. *CACM 20*, 8 (1977), 553-564.
14. Stroustrup, B. *The C++ Programming Language*. Addison-Wesley, 1986.