Chapter 27

Implementation

It is not the intent of this book to provide a detailed introduction to programming language implementation. Nevertheless, a general understanding of the problems encountered in implementing object-oriented languages, and the various ways to overcome them can, in many cases, help the reader better understand object-oriented techniques. In particular, this will help clarify the way in which object-oriented systems differ from more conventional systems. In this chapter, we provide an overview of some of the more important implementation techniques, as well as pointers to the relevant literature for the reader who desires further information.

27.1 Compilers and Interpreters

Broadly speaking, there are two major approaches to programming language implementation: compilers and interpreters. A *compiler* translates the user's program into native machine code for the target machine and is invoked as a separate process independent of execution. An *interpreter*, on the other hand, is present during execution, and is the system that runs the user program.

As is true of most distinctions, while the endpoints are clear there are large gray areas in the middle. There are compilers that compile interactively even during executing (at least during the debugging stages). These compilers gain some of the advantages of the interpreter while giving the execution-time advantage of the compiler technique. Similarly, some interpreters can translate into either an intermediate representation or native code.

Generally, a program that is translated by a compiler will execute faster than will a program that is run under an interpreter. But the time between conception, entering text, and execution in a compiled system may be longer than the corresponding time in an interpreter. Furthermore, when errors occur at run time, the compiler often has little more than the generated assembly language to offer as a marker to the probable error location. An interpreter will

usually relate the error to the original text the user entered. Thus, there are advantages and disadvantages to both approaches.

Although some languages are usually compiled and others are usually interpreted, there is nothing intrinsic in a language that forces the implementor to always select one over the other. C++ is usually compiled, but there are C++ interpreters. On the other hand, Smalltalk is almost always interpreted, but experimental Smalltalk compilers have been produced.

27.2 The Receiver as Argument

In a compiled language, ultimately all methods are translated into functions that are, in many respects, just like any other function. The instructions for the function are rendered as a sequence of assembly language instructions which reside in a fixed location in memory, and when the function is executed control is transferred to this location. As part of this transfer of control, an activation record[1] is created to hold the parameters and the local variables. So how does this code gain access to the instance data associated with the receiver?

To put the question in concrete terms, recall the class description of CardPile from the solitaire program:

```
class CardPile {
public:
    ...
        bool addCard (Card * aCard);
    ...
private:
    list<Card *> cards;
    const int x;
    const int y;
};

void CardPile::addCard (Card * aCard)
{
    card.push_front(aCard);
}
```

The compiler creates assembly language code for the method addCard. Seemingly, the only parameter is the playing card named aCard. How does this code gain access to the data field card?

[1] The activation record is a portion of the run-time stack set aside at procedure entry to hold parameters, local variables, and other information. Further details on the run-time environment of programs can be found in any compiler-construction textbook. See the section on further reading at the end of the chapter.

The answer is that the receiver is in reality passed as a hidden first parameter. An invocation, such as:

```
CardPile * aCardPile = ...;
Card * currentCard = ...;

aCardPile->addCard (currentCard);
```

is in reality translated as if it had been written:

```
addCard(aCardPile, currentCard);
```

At the other end, the code for the method is compiled as if it had been written as follows:

```
void addCard (CardPile * this, Card * aCard)
{
    this->card.push_front(aCard);
}
```

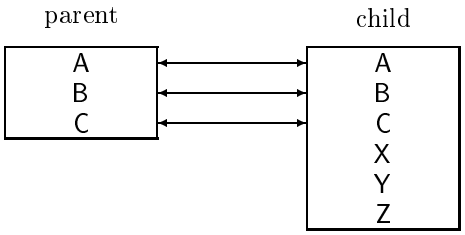
Notice how the pseudo-variable this has become a real first parameter. References to data fields or to methods within the class can be handled using this value.

27.3 Inherited Methods

The next problem we will consider is how it is possible for a method defined in a parent class to continue to function even when it is executed using an instance of a child class as receiver. This question arises both for differences in the receiver and differences in argument caused by the use of polymorphic variables as parameters. Indeed, as was noted in the previous section, the receiver for a message passing expression *is* just a polymorphic argument.

To understand how unusual this is, note that in most other ways argument values can never be changed. It would not be possible for a procedure that is expecting an integer as an argument to work correctly when it is given a string.

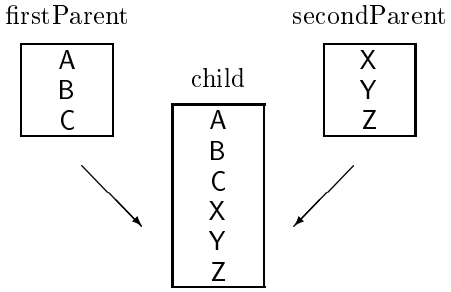
The key insight that allows inherited methods to continue to operate is the way that child classes are represented in memory. In a compiled version of a program data fields are not accessed by name, but by a fixed offset relative to the beginning of an object. A child class will store the same fields at the same offsets. They may add new data fields, but these will be as *extensions* of the parent data fields.



Since the code generated when the parent class is processed only requires that data fields be found at a known offset, the code will work regardless of whether it is dealing with an instance of the parent or an instance of the child.

27.3.1 The Problem of Multiple Inheritance

One of the reasons why multiple inheritance is difficult to implement is precisely because a child class cannot store inherited data fields in exactly the same location as they are found in *both* parents. The child can mirror one parent or the other, but not both at the same time.



There are various solutions to this problem, but none result in a system that is as simple as that provided by single inheritance. The most common technique is that an object pointer is represented as a pair containing both the pointer to the base and an offset. However, the details concerning how this can be done are beyond the scope of this book.

27.3.2 The Slicing Problem

That a child class can only *extend* the data fields defined by a parent class nicely solves the problem of inherited methods. That is, it gives the compiler a way to generate code for procedure defined in a parent class so that it will nevertheless work on objects of a child class. But this same property introduces another problem.

As we discussed in earlier chapters, a *polymorphic variable* is one that is declared as representing one type, but which in fact holds values from another type. In object-oriented languages the values usually must come from a subclass of the parent class.

When a compiler sets aside space in an activation record, it generally knows only the declared type for a variable, not the run-time type. The question is therefore how much memory should be allocated in the activation record. As we discussed in Chapter 12, most programming languages elect one of two solutions to this problem:

- The activation record holds only pointers, not values themselves
- The activation record holds only the data fields declared in the parent, slicing off any data fields from the child class that will not fit.

There are merits to both alternatives, so we will not comment on which technique seems "better." However, as a programmer it is important that you understand the technique used by the system on which you work. C++ uses the slicing approach, Java and most other object-oriented languages use the pointer approach.

27.4 Overridden Methods

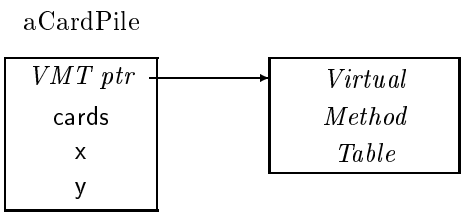
We have explained how inherited methods can execute even when presented with an instance of a child class, but what about the inverse? How is it possible that when an overridden method is invoked the code that is executed will be that associated with the current value of the receiver, regardless of its declared type?

To put the question in concrete terms, recall that in our solitaire game the method addCard is redefined in the child class DiscardPile. A variable that is declared as maintaining a CardPile will, if it actually references a DiscardPile, execute the correct method:

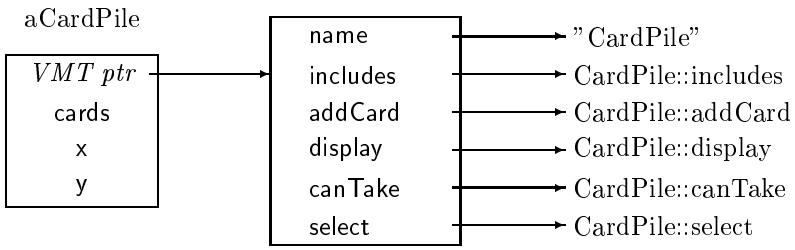
```
CardPile * aCardPile = new DiscardPile();
Card * aCard = ...;

aCardPile->addCard (aCard); // will execute DiscardPile::addCard
```

Since the dynamic class can change during execution, there must be something stored in the variable that indicates the type of value it is currently maintaining. This value is termed a *virtual method table pointer*. It is simply an additional data field, a hidden pointer that references an object called the *virtual method table*:



The virtual method table is a static data area constructed for each class. All instances of the same class will point to the same virtual method table. In some implementations the virtual method table may include a small amount of useful information, such as the class name or the size of instances of the class in bytes, but the most important part of the virtual method table is an array of pointers to functions.



The offset to any particular method can be determined at compile time. Thus, an invocation of a virtual method is translated into an indirect access mediated through the virtual method table. The call:

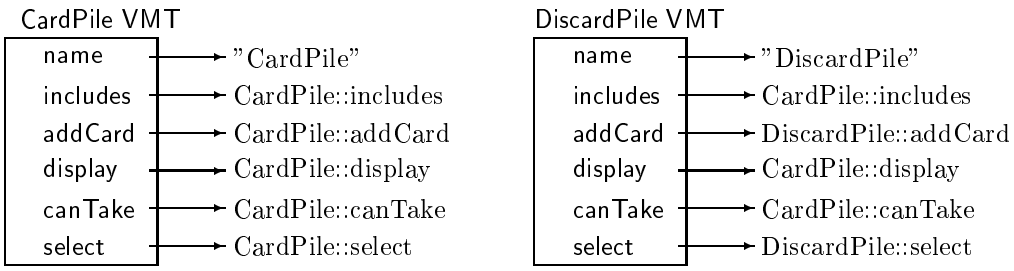
```
aCardPile->addCard (aCard);
```

becomes

```
aCardPile->VMTptr[3](aCardPile, aCard);
```

where VMTptr is the name of the hidden data field that references the virtual method table, and the offset of the addCard method is assumed here to be 3. (Note we have also passed the receiver as argument, as we described in Section 27.2).

The next part of the solution to the problem of overridden functions deals with the layout of the virtual method table for a child class in relation to the table for the parent class. In short, overridden methods are placed at the same offsets in both, but point to different functions:



If a method is not overridden, the pointer in the virtual method table will be the same in the child as in the parent. If it is overridden, the location in the child

will point to the child code, while the location in the parent will point to the parent code. New methods defined in the child but not found in the parent are tacked on to the end of the childs virtual method table. Thus the table for the child is an extension of the parent, in much the same way that the data layout for the child is an extension of the data layout for the parent.

In this way the invocation of of an overridden method can quickly and easily be resolved to the correct function. Notice that the overhead for a virtual method call is one level of indirection (the reference to the virtual method table) and one array index (the index into the table). On most machines this can be accomplished in one or two assembly language instructions.

27.4.1 Eliminating Virtual Calls and Inlining

Although the overhead of a virtual method invocation in comparison to a normal function call is small, in some instances (for example inside of loops) even a small difference can be critical. If a compiler can determine at compile time the dynamic class of a receiver, then an invocation of a virtual method can be transformed into a normal procedure call, avoiding the overhead of the virtual method table lookup.

Most often this is accomplished through a technique termed data flow analysis. A careful trace is performed of all execution paths between the point a variable is given a value and the point it is used as a receiver. In many cases this analysis will reveal the exact type for the variable.

The elimination of virtual method calls is often combined with *method inlining*. Object-oriented programming tends to encourage the development of many small methods; often much smaller than the average function in an imperative language. If data flow analysis can link a message invocation to a specific method, and if the method is very small (for example, it may simply return a data field), then the body of the method can be expanded in-line at the point of call, thereby avoiding the overhead of the procedure call.

27.5 Name Encoding

Since methods are all known at compile time and cannot change at run time, the virtual tables are simply static data areas established by the compiler. These data areas consist of pointers to the appropriate methods. Because linkers and loaders resolve references on the basis of symbols, some mechanism must be provided to avoid name collisions when two or more methods have the same name. The typical scheme combines the names of the class and the method. Thus, the addCard method in class DiscardPile might internally become DicardPile::addCard. Usually, the user need never see this name, unless he is forced to examine the assembly-language output of the compiler.

In languages such as C++ that allow methods to be further overloaded with

disambiguation based on parameter type, even more complicated Gödel-like[2] encodings of the class name, method name, and argument types are required. For example, the three constructors of class Complex described in an earlier chapter might be known internally as Complex::Complex, Complex::Complex_float, and Complex::Complex_float_float, respectively. Such internal names, sometimes referred to as *mangled names*, can become very long. As we have seen, this internal name is not used during message passing but merely in the construction of the virtual tables and to make unique procedure names for the linker.

27.6 Dispatch Tables

Because languages such as C++ and Object Pascal are statically typed, they can determine at compile time the parent class type of any object-oriented expression. Thus, a virtual method table needs to be only large enough to accommodate those methods actually implemented by a class. In a dynamically typed language, such as Smalltalk or Objective-C, a virtual method table has to include *all* messages understood by any class, and this table needs to be repeated for every class. If an application has 20 classes, for example, and they each implement 10 methods on average, we need 20 tables, each consisting of 200 entries. The size requirements quickly become exorbitant, and a better technique is called for.

An alternative technique is to associate with every class a table that, unlike the virtual method table, consists of selector-method pairs. This is called a *dispatch table*. The selectors correspond only to those methods actually implemented in a class. Inherited methods are accessed through a pointer in this table, which points to the dispatch table associated with a superclass (see Figure 27.1).

As in a system using virtual method tables, when dispatch tables are used every object carries with it an implicit (that is, not declared) pointer to the dispatch table associated with the class of the value it represents. This implicit pointer is known as the isa link (not to be confused with the *is-a* relation between classes). A message expression in Objective-C, such as the following expression from the eight-queens problem:

```
[neighbor checkrow: row column: column]
```

is translated by the Objective-C compiler[3] into:

```
objc_msgSend(neighbor,"checkrow:column:", row, column)
```

[2] The term "Gödel-like" refers to the technique of encoding a large amount of information (such as an entire computer program) as a single quantity. The technique was first described by the German computer scientist Kurt Gödel in a paper in 1931 [Gödel 1931]. Its use in a linker was, to my knowledge, first described by Richard Hamlet [Hamlet 1976].

[3] The Objective-C system is a translator that produces conventional C code. In addition, the string form of the selector is not actually used; instead, selectors are hashed into a numeric value.

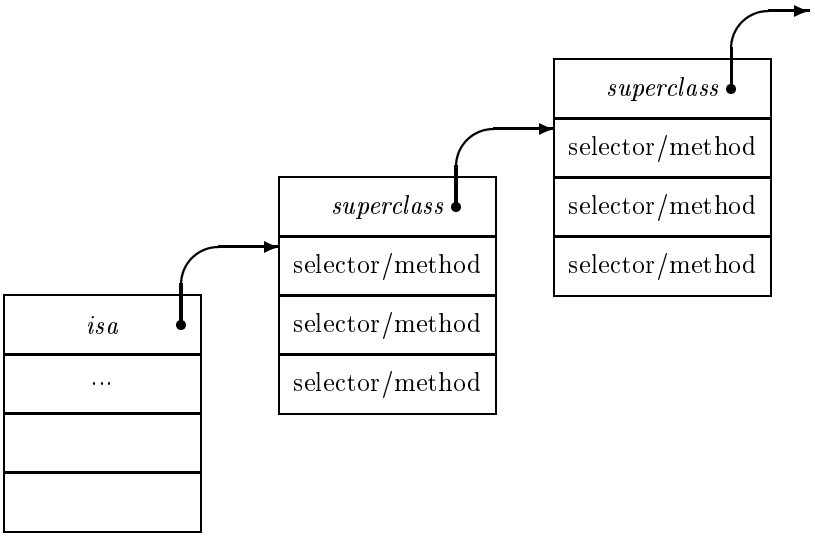


Figure 27.1: – An object and its dispatch table.

The function objc_msgSend, called the *messaging function*, follows the *isa* link of the first argument to find the appropriate dispatch table. The messaging function then searches the dispatch table for an entry that matches the selector. If such an entry is found, the associated method is invoked. If no such method is found, the dispatch table of the superclass is searched. If the root class (class Object) is finally searched and no method is found, a run-time error is reported.

27.6.1 A Method Cache

Although, for dynamically typed languages, the dispatch table is more economical in space than the virtual method table, the time overhead is considerably greater. Furthermore, this overhead is proportional to the depth of inheritance. Unless this penalty can be overcome, the latter point might lead developers to abandon inheritance, trading the loss in power for the gain in efficiency.

Fortunately, we can largely circumvent this execution-time loss by means of a simple technique. We maintain a single system wide *cache* of methods that have been recently accessed. This cache is indexed by a hash value defined on the method selectors. Each entry in the cache is a triple, consisting of a pointer to a class (the dispatch table itself can serve this purpose), a selector value, and a pointer to a method.

When the messaging function is asked to find a method to match to a selector class pair, it first searches the cache (see Figure 27.2). If the entry in the cache at the hash-table location corresponds to the requested selector and class, the

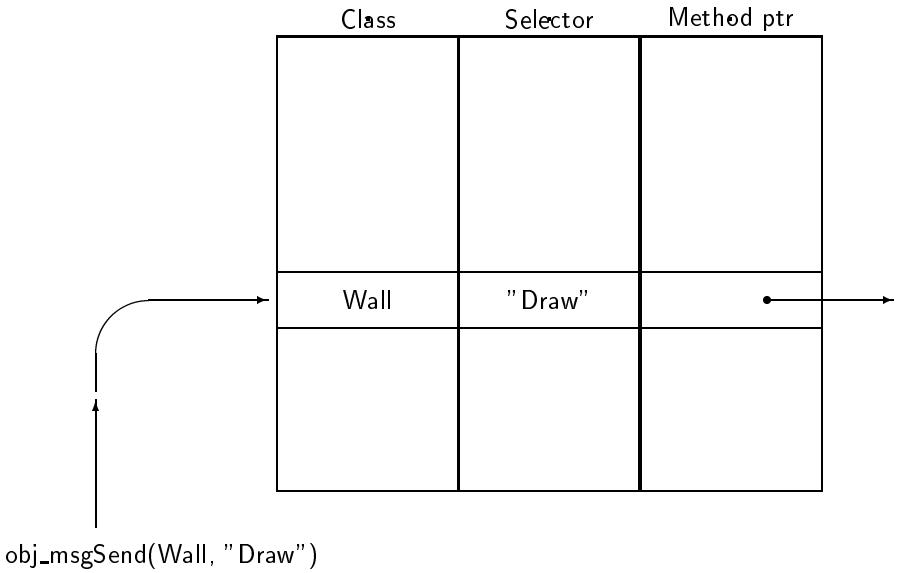


Figure 27.2: – The messaging function checking the method cache.

associated method can be executed directly. If not, the search process described earlier is performed. Following this search, immediately before executing the method the cache is updated, overwriting whatever entry it contained previously at the hash location given by the message selector. Note that the value stored for the class entry in the cache is the class where the search began, not the class in which the method was eventually discovered.

By appropriate selection of hash functions and cache sizes, one can achieve cache hit ratios in the range of 90 to 95 percent, which reduces the overhead involved in a messaging expression to slightly over twice that of a conventional procedure call. This figure compares favorably with the overhead incurred with the virtual method table technique.

27.7 Bytecode Interpreters

Interpreters are usually preferred over compilers if the amount of variation in a program is larger than can be accommodated easily in fixed code sequences. This variation can come from a number of sources; in a dynamically typed language, for example, we cannot predict at compile time the type of values that a variable can possess (although Objective-C is an example of a dynamically typed language that is nevertheless compiled). Another source of variation can occur if the user can redefine methods at run time.

A commonly used approach in interpreters is to translate the source program into a high-level "assembly language," often called a *bytecode* language (because

0000xxxx	Extended instruction with opcode xxxx
0001xxxx	Push instance variable xxxx on stack
0010xxxx	Push argument xxxx on stack
0011xxxx	Push literal number xxxx on stack
0100xxxx	Push class object number xxxx on stack
0101xxxx	Push system constant xxxx
0110xxxx	Pop into instance variable xxxx
0111xxxx	Pop into temporary variable xxxx
1000xxxx	Send message xxxx
1001xxxx	Send message to super
1010xxxx	Send unary message xxxx
1011xxxx	Send binary message xxxx
1100xxxx	send arithmetic message xxxx
1101xxxx	Send ternary message xxxx
1110xxxx	Unused
1111xxxx	Special instruction xxxx

Figure 27.3: – Bytecode values in the Little Smalltalk system.

typically each instruction can be encoded in a single byte). Figure 27.3 shows, for example, the bytecode instructions used in the Little Smalltalk system. The high-order four bits of the instruction are used to encode the opcode, and the low-order four bits are used to encode the operand number. If operand numbers larger than 16 are needed, the extended instruction is used and the entire following byte contains the operand value. A few instructions, such as "send message" and some of the special instructions, require additional bytes.

The heart of the interpreter is a loop that surrounds a large switch statement. The loop reads each successive bytecode, and the switch statement jumps to a code sequence that performs the appropriate action. We will avoid a discussion of the internal representation of a program (interested readers are referred to [Budd 1987]) and will concentrate solely on the processing of message passing.

```
while (timeslice-- > 0) {
  high = nextByte();  // get next bytecode
  low = high & 0x0F;  // strip off low nybble
  high >>= 4;     // shift left high nybble
  if (high == 0) {  //check extended form
    high = low;  // if so use low for opcode
    low = nextByte();  // get real operand
    }

  switch(high) {
    case PushInstance: ...
    ...
    case PushArgument: ..
    ...
    }
  }
```

Just as objects in the compiled system presented earlier all contain a pointer to a virtual table, objects in the Smalltalk system all contain a pointer to their class. The difference is that, as we saw in Chapter 4, the class is itself an object. Among the fields maintained in the class object is a collection containing all the methods corresponding to messages that instances of the class will understand (Figure 27.4). Another field points to the superclass for the class.

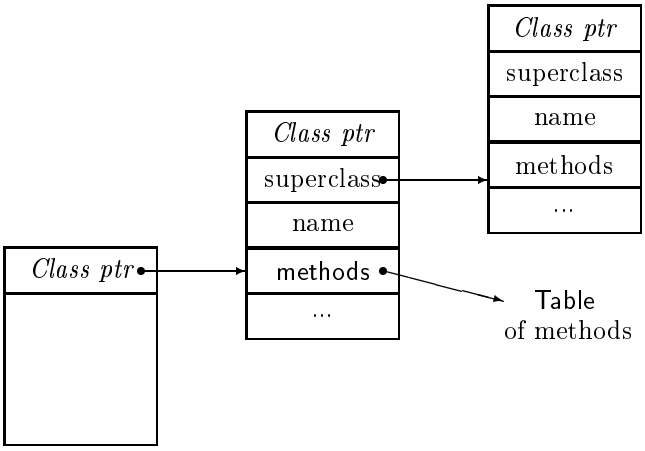


Figure 27.4: – The internal structure of a class.

When a message is to be sent, the interpreter must first locate the receiver for the message. By examining the class pointer for the receiver, it can find the

object corresponding to the class of the receiver. It then searches the methods collection for a method that matches the name of the message being sent. If no such method is found, it follows the superclass chain, searching the classes in the superclass until either an appropriate method is found or the chain is exhausted. In the latter case, the interpreter reports an error. This is exactly the same sequence of steps as performed by the messaging function used in the dispatch table technique. As with that technique, a cache can be used to speed up the process of method search.

Bytecode interpreters have recently been popularized through their use by Java systems. While the range of bytecodes used by Java is much more extensive than what has been described here, the key ideas remain the same.

27.8 Just-In-Time Compilation

The key criticism usually leveled against interpretive systems is that there execution time is typically much slower than that obtained from compilers. On the other hand, interpreters can be much more portable. Java is the language that has most recently trumpeted this advantage, claiming that Java bytecodes created on any machine can be executed on any other machine.

A scheme that tries to balance the benefits of both techniques is *just in time* compilation, or JIT. In a JIT system a program is first translated into a portable, high level form; for example Java bytecodes. When these bytecodes get loaded onto a specific machine, then at that instance, just before execution (the "just in time" of the name), the bytecodes are translated once again into machine code for the hardware on which execution is taking place.

The JIT technique gives the advantage of portability (the original bytecode form of the program can be moved to any machine) with the execution time performance of compiled code. There is a small time penalty that must be paid for translating the bytecodes into machine code; but normally the resulting machine code is saved so that if the same method is executed repeatedly this translation cost is incurred only once and can be amortized over all the successive calls.

The major difficulty with JIT systems is that they are complicated systems to develop, particularly if they are combined with optimization techniques. But as the dramatic improvements in execution speeds of Java systems in the past few years has indicated, the benefits can be impressive.

Chapter Summary

Regardless whether an implementation is provided by a compiler or an interpreter, there are several fundamental problems that must be addressed. In this chapter we have examined the following:

- Providing access to the receiver from within a method.

- How an inherited method can continue to operate using an instance of a child class in place of an instance of the parent class.
- How methods can be overridden, and how a dynamic dispatch technique for a statically typed languages can select at run time the method that will match a polymorphic variables dynamic class.
- How unique names can be created for system software, such as linkers, that require this property.
- How dynamic method lookup is performed in dynamically typed languages.
- How bytecodes can be used as a flexible and portable internal representation of an object-oriented program.
- How Just-In-Time compilation can provide the benefits of both portability and execution efficiency.

Further Reading

A good introduction to the problems of language implementation can be found in compiler construction textbooks, such as [Aho 1985, Fischer 1988].

For the reader interested in learning more about the implementation of object-oriented languages, Cox [Cox 1986] contains a detailed analysis of the time-space trade-offs involved in various schemes. The implementation of multiple inheritance in C++ is sketched in [Ellis 1990], which is based on an earlier algorithm for Simula [Krogdahl 1985]. A detailed description of C++ implementation techniques is provided by Lippman [Lippman 1996].

The Smalltalk-80 interpreter is described in [Goldberg 1983]. [Krasner 1983] contains several papers that describe techniques for improving the efficiency of the Smalltalk-80 system. A simplified Smalltalk interpreter is detailed in [Budd 1987]. Kamin [Kamin 1990] presents a good general overview of the issues involved in the implementation of nontraditional languages.

Self Study Questions

1. In broad terms, what are the differences between compilers and interpreters? What are some advantages of each technique?
2. When a method is translated into an ordinary function, what changes are necessary in order to provide access to the receiver?
3. What key feature of the memory layout of objects permits methods defined in a parent class to be used with instances of a child class?
4. Why is the simple approach the inherited methods that is used by single inheritance languages not applicable when multiple inheritance is allowed?

5. Describe the run-time technique used to match the invocation of an overridden method to the correct method implementation.

6. What is a mangled name? Why is it necessary to create mangled names?

7. In what ways is a dispatch table different from a virtual method table? In what ways are they similar?

8. Explain how a message selector is matched to a method when the dispatch table technique is used.

9. What is a method cache? How does it speed up the task of message passing?

10. What is a bytecode?

11. Why are JIT systems described as "just in time"?

Exercises

1. Extend the dispatch table technique to permit multiple inheritance.

2. The Objective-C compiler permits optional declarations for object variables. Explain how a compiler might make use of such declarations to speed processing of messages involving such values. Consider what needs to occur on assignment and how messaging can be made more efficient.

3. Explain why methods that are not declared virtual in C++ can be invoked more efficiently than can virtual methods. How do you make measurements to determine whether the difference is significant?

4. Review the cache technique described in Section 27.6.1. Explain why the class stored in the cache is the one where the search for a method begins and not the one where the method is eventually found. Explain how the cache lookup algorithm would need to be changed if the latter value were used. Do you think the new algorithm would be faster or slower? Explain your answer.

5. Sketch the outline of a Smalltalk interpreter based on the bytecodes given in the text.