Capsules

Richard Zippel
Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Mass. 02139

Well organized large systems tend to consist of a large number of small pieces of code each of which captures a single semantic unit. These pieces of code are strung together to form larger semantic phrases, which are in turn components of even larger phrases. The smallest semantic units in a polynomial manipulation system might be the routines that add and multiply the coefficients of the polynomials. These routines are combined (used as subroutines) to form the routines that add and multiply polynomials, which are components in the factoring and greatest common divisor routines.

When a system is built in a top down manner, the larger phrases are formed first and are used to define the semantic components of the smaller phrases. Bottom up software design begins with the small phrases and generates the large phrases. In practice a combination of these two approaches is often used. The manner in which the routines are initially connected is usually simple, but in time the addition of new capabilities and features, and the necessities of performance enhancement generally cause the dependency structure to become quite complex.

A good example of this sort of complexity is when a new, "higher performance" representation of some data structure is introduced for critical uses (caching and hash tables examples of this optimization). In building an input/output system for a computer, we might initially specify a *stream* to be a simple, character at a time structure. When this structure is used for file operations and networking it becomes necessary to add buffering, but it would be unwise to use the buffered stream for terminal I/O. These two types of streams could share large amounts of code if the manner in which the lower level routines are "glued together" is sufficiently powerful. For instance, the only difference between the routines which close the stream is that the buffered stream must flush its buffers and return them to the buffer pool.

© 1983 ACM 0-89791-108-3/83/006/0166 $00.75

We are particularly interested in the "gluing together" process which is used to form large systems. In this paper we will describe a system called *Capsules* which we feel provides a more natural and more powerful combination mechanism than discussed previously. This system was originally an attempt to simplify the construction of an algebraic manipulation system, but we are now applying it to the development of a VLSI design system and investigating its utility in organizing the I/O system of a complex personal computer.

1. Philosophy

In most systems, when a piece of code is written it is given a name. In the earliest programming languages (Fortran, Basic, Lisp 1.5), When the user wants to perform some operation (like pushing an element on a stack or outputting a character), it is necessary to find a piece of code that implements the desired operation and refer to it using its name. In some systems (CLU [Lis77], Flavors [Wei81], Loops [Bob82], Smalltalk [Ing76, Xer81]) an extra level of indirection is introduced that allows the binding of the piece of code to an operation name to be delayed until after the code is written. This approach has been called *data abstraction*. In compile-time languages like CLU, the association of the code with the abstract operation is made at compile or link time. The Lisp and Smalltalk versions of these approach delay the binding until runtime. In either case, an extra level of indirection has been provided between the name representing an abstract operation and the piece of code that implements that abstraction. There is still no tie between the abstract operation as a semantic unit that the user wants to use and the piece that implements the operation.

In the Capsule system, the user specifies the desired behavior of the operation and the system is responsible for finding the piece of code that implements that operation and is compatible with previous constraints. If a more efficient piece of code is written that implements some operation, the system will use the more efficient code as long as it meets the user's specifications. This is a result of (1) the user referring to code fragments by their semantic purpose rather than their name, and (2) the system being responsible for matching the the semantic requests with the code fragments in the system.

2. Capsules

In the capsule system we have assumed that all actions occur by sending messages to objects—this is the *object oriented viewpoint*. Taking the dual point of view, where objects are passive and the correct function is chosen by the compiler, merely moves the mechanism we are discussing into the compiler. This is the fundamental difference between the Lisp Machine's flavor system which takes the object oriented viewpoint, and CLU which is function oriented.

By an *object* we will mean something to which a *message* can be sent. This will result in one (or more values) being returned and the internal state of the object being changed. The action caused when a message is sent to an object is called an *operation*. A message is a string used as the name of some operation. It contains no internal structure. The piece of code executed when a message is sent to an object is called a *method*. Objects may also contain internal state which is kept in *instance variables* that may be referenced by the methods.

Every object belongs to a class of equivalent objects that have the same methods and the same set of instance variables. This equivalence class is called a *collage*. Objects are created by calling the function MAKE-OBJECT on a collage. The methods are actually part of the collage, so as operations are added to the collage, the objects of the collage also acquire them.

The specification for how a method is to be constructed is kept in a structure called a *capsule*. When a capsule is added to a collage, the code within the capsule is incorporated in one or more of the methods of the collage. Capsules also contain information describing what their pieces of code expect of the collage to which they are added (what operations and instance variables there are, for instance).

The design of the capsule system was based on our experience with some very large software systems, and it is in the construction of large systems that its power is most apparent. The following paragraphs use a small example to explain the mechanisms and terminology of the Capsule system. As such, the Capsule mechanisms may seem to be overkill. The reader is asked to treat this small example as what it is, and to map the capsule mechanisms onto whatever large software system is familiar.

The small example we use is the implementation of a stack. A stack is an object that accepts two messages, PUSH and POP. These messages have the obvious meaning. We will additionally introduce an operation called TWIDDLE which interchanges the top two elements of a stack. Two implementations are given for stacks. One uses a list to implement the stack, while the other uses an array.

An *operation* is the specification of an action. It includes specifications for the number and type of arguments and return values as well as a specification of what the action will accomplish, called the *semantics* of the operation. The current implementation does not interpret the semantics in any manner. The semantics fields are checked for equality to ensure that two operations perform the same action.

To implement our stack example, the first thing we need to do is specify the operations that will be used, PUSH, POP and TWIDDLE.

```
    (DEFOPERATION PUSH
      (ARGUMENTS NIL)
      (RETURNS)
      (SEMANTICS PUSH-ELEMENT-ON-STACK)
      (DOCUMENTATION "Adds an element to the top
of a stack"))
```

The ARGUMENT field indicates that PUSH takes exactly one additional argument, its type is unspecified. No values are returned. The semantics field has the atom PUSH-ELEMENT-ON-STACK in it. Since the the semantics field is not really interpreted by the current version of this system, this atom is used as a place holder. We will leave out the semantics fields in the following examples. The documentation string is used by the run-time documentation system.

```
    (DEFOPERATION POP
      (RETURNS NIL)
      (DOCUMENTATION "Removes and returns the top
element of a stack"))

    (DEFOPERATION TWIDDLE
      (DOCUMENTATION "Exchanges the top two ele-
ments of a stack"))
```

The default assumptions are that an operation takes no arguments and returns no values. These assumptions are used in the specifications of POP and PUSH.

Protocol's are used to specify the characteristics of a collage. A protocol is a list of (1) operations (including their semantics), (2) axioms, which specify relationships among operations, (3) instance variables, and (4) attributes, which are other characteristics. The following protocol captures the notion of a stack.

```
    (DEFPROTOCOL BASIC-STACK
      (OPERATIONS PUSH POP)
      (AXIOM
       (STACK-PUSH-POP-AXIOM PUSH POP)))
```

That is, a stack accepts two operations, PUSH and POP (as described above) and these two operations obey the STACK-PUSH-POP-AXIOM, which means that a PUSH followed by a POP returns the value originally pushed and all the inductive variations of that statement. As with the semantics portions of operation specifications, the current system does not attempt to interpret the axioms in more primitive terms, but treats them atomically.

The following slightly more complex protocol illustrates how the mathematical abstraction of an algebraic ring may be specified.

```
(DEFPROTOCOL RING
  (OPERATIONS PLUS MINUS ZERO
              TIMES)
  (AXIOMS
   (COMMUTATIVE-LAW PLUS)
   (ASSOCIATIVE-LAW PLUS)
   (ASSOCIATIVE-LAW TIMES)
   (ALGEBRAIC-IDENTITY PLUS ZERO)
   (ALGEBRAIC-INVERSE PLUS MINUS)
   (DISTRIBUTIVE-LAW PLUS TIMES))
  (ATTRIBUTES
   (CHARACTERISTIC)))
```

It is assumed that the DEFOPERATIONs for the specified operations appear elsewhere. This specification indicates that there is a ZERO operation that returns the additive identity. An alternative implementation might require ZERO to be an instance variable. Any collage that adheres to this protocol is an *abstract ring*. Any piece of code that depends only on this protocol can be added to an abstract ring. This is somewhat closer to the mathematical understanding of abstraction than previous systems.

The DEFOPERATION form defines a protocol that contains a single operation and nothing else. The name of this protocol is the same as the name of the operation unless specified otherwise. This conveniently allows the use of operation names and protocols interchangeably.

All code is put into capsules. Capsules consist of four basic parts, (1) a required protocol, what the capsule expects of the collage it is to be added to, (2) an asserted protocol, things to add to the protocol of a collage when the capsule is added, (3) performance information about the algorithm contained in the capsule, and (4) the code itself. It often happens that more than one capsule could be added to a collage to satisfy some requirement. The performance information is used to break those deadlocks.

In order to allow incremental compilation and debugging, the specification of a capsule is separated into two pieces. A DEFCAPSULE form is used to indicate the first three parts of the specification while separate DEFALGORITHM forms are used for each piece of code. (In the terminology used in the flavor system, capsules are extensions of flavors, and algorithms are methods. Our algorithms can be more complex than the simple pieces of code that flavor methods must be, but it would take us too far afield to discuss these capabilities here.) The following implementation of the BASIC-STACK protocol illustrates this.

```
(DEFCAPSULE LIST-STACK
  (ASSERTS
   (PROTOCOL BASIC-STACK)
   (INSTANCE-VARIABLE (STACK ()))
   (ATTRIBUTE STACK-IMPLEMENTED-AS-LIST)))

(DEFALGORITHM (LIST-STACK PUSH) (ELEMENT)
  (SETQ STACK (CONS ELEMENT STACK)))

(DEFALGORITHM (LIST-STACK POP) ()
    :OG1 (FIRST STACK)
         (SETQ STACK (REST STACK))))
```

The LIST-STACK capsule implements a stack in terms of a list of the elements of the stack. It makes no assumptions of the collage to which it is to be added. When it is added to a collage, the BASIC-STACK protocol is added, along with an instance variable STACK and the two pieces of code given in the DEFALGORITHM. In addition, an attribute is added to the collage that indicates the stack is implemented using a list.

The capsule that implements stacks in terms of arrays, is quite similar (we have ignored the problem of running off the end of the array for simplicity here).

```
(DEFCAPSULE ARRAY-STACK
  (ASSERTS
   (PROTOCOL BASIC-STACK)
   (INSTANCE-VARIABLES
     (STACK (MAKE-ARRAY '(100)))
     (INDEX 0))
   (ATTRIBUTE STACK-IMPLEMENTED-BY-ARRAY)))

(DEFALGORITHM (ARRAY-STACK PUSH) (ELEMENT)
  (SETF (AREF STACK INDEX) ELEMENT)
  (SETQ INDEX (+ INDEX 1)))

(DEFALGORITHM (ARRAY-STACK POP) ()
  (SETQ INDEX (- INDEX 1))
  (AREF STACK (+ INDEX 1)))
```

The function MAKE-COLLAGE is used to create collages. It takes an arbitrary number of arguments, each of which is either a capsule or the name of a capsule, or a protocol or the name of a protocol. Thus the following forms can be used to construct stack collages of the two type defined thus far.

```
(SETQ C1 (MAKE-COLLAGE 'LIST-STACK))
(SETQ C2 (MAKE-COLLAGE 'ARRAY-STACK))
```

The form (MAKE-COLLAGE 'BASIC-STACK) would result in an error because it is ambiguous. There are two capsules that can be used to create a collage with the BASIC-STACK protocol and there is no reason to prefer one over the other. (In this situation we have seriously considered just picking the first capsule. The user hasn't given any reason to prefer one capsule over the other so why not pick one at random?)

Once we have a couple of collages to work with, we can create stacks using the MAKE-OBJECT function. Its first argument is either a collage or the name of one. Additional arguments are passed to the initialization method if there is one. The following forms, create a stack from the collage C1 and push two elements onto it.

```
(SETQ STACK (MAKE-OBJECT C1))
(SEND STACK 'PUSH 1)
(SEND STACK 'PUSH 2)
```

Now, (SEND STACK 'POP) will return 2.

Though we have defined what is meant by the TWIDDLE operation, no capsule implements it. The following capsule provides an algorithms that "TWIDDLEs" the top two elements of an abstract stack.

```
(DEFCAPSULE BASIC-TWIDDLE
  (REQUIRES BASIC-STACK)
  (PERFORMANCE 1)
  (ASSERTS
   (PROTOCOL TWIDDLE)))

(DEFALGORITHM (BASIC-TWIDDLE TWIDDLE) ()
  (LET (TOP SECOND)
    (SETQ TOP (SEND SELF 'POP)
          SECOND (SEND SELF 'POP))
    (SEND SELF 'PUSH TOP)
    (SEND SELF 'PUSH SECOND)))
```

This capsule has a required protocol, BASIC-STACK. Thus it can only be combined with collages that already possess the PUSH and POP operations. It can be added to any abstract stack.

The routine ADD-PROTOCOLS is used to add protocols to collages. Its first argument is a collage and the rest of its arguments are protocols that the user wants the collage to meet. Thus the form

```
(ADD-OPERATION C1 'TWIDDLE)
```

adds a TWIDDLE operation to C1. More precisely, each collage contains a table that gives the relationships between message names and pieces of code. Each object constructed from a collage contains a pointer to this table. When an operation is added to a collage, the system isolates a capsule that both provides the desired operation and which can be added to the collage. The code portion of the capsule is then added to the collage's method table. Thus all objects of the collage are now extended with the new operation.

It is easy to define a slightly more efficient version of TWIDDLE for arrays. The ARRAY-TWIDDLE capsule does precisely this.

```
(DEFCAPSULE ARRAY-TWIDDLE
  (REQUIRES
    (ATTRIBUTE STACK-IMPLEMENTED-BY-ARRAY))
  (PERFORMANCE 2)
  (ASSERT
    (PROTOCOL TWIDDLE)))
(DEFALGORITHM (ARRAY-TWIDDLE TWIDDLE) ()
  (LET ((TEMP))
    (SETQ TEMP (AREF STACK (- INDEX 1)))
    (SETF (AREF STACK (- INDEX 1))
          (AREF STACK (- INDEX 2)))
    (SETF (AREF STACK (- INDEX 2)) TEMP)))
```

Notice that this capsule does not actually use the PUSH and POP operations. It only assumes that there are instance variables STACK and INDEX, and that they can be interpreted to form a stack. This is the purpose of the STACK-IMPLEMENTED-BY-ARRAY attribute.

With this capsule added to the system, adding the TWIDDLE operation to an ARRAY-STACK collage will get the new code, while previously it would have used the routine in BASIC-TWIDDLE.

As a final note, if a message is sent to an operation that does not possess a handler for that message then a *default-handler* is run. One of the default handlers with which we have been experimenting attempts to add the desired operation to the object's collage and then tries again. Thus if a stack did not have a TWIDDLE handler, a TWIDDLE message could be sent to it anyway, since one could be created for it and installed on the fly. If the stack was an ARRAY-STACK then the efficient ARRAY-TWIDDLE capsule would be added, otherwise the BASIC-TWIDDLE can be used.

There are two points to notice about this scenario. First, when new functionality was added to a collage, the user specified only what the desired semantics were and did not specify, directly or indirectly, a particular piece of code. Second, the functionality could be added dynamically while the system is running. In some domains, there are several algorithms for performing an operation and it can be very expensive to decide which to use. It is better not to pay that price until it is truly necessary.

3. Conclusions

When the capsule system was used to describe a portion of the stream code used in the LISP Machine, we noticed that the protocol specifications seemed more verbose than we would have liked. Closer examination revealed that many of the comments we had penciled into the version of the code that used flavors were being translated into protocols. This reinforces our impression that the capsule system is partially an attempt to force the programmer to make the code that is written more precise.

We feel that if the programmer makes this effort, the programming system will be in a much better position to aid in the development of large software systems. The Capsule system is a partially successful attempt to provide mechanism through which the programmer can truly express what the program is intended to do.

4. Acknowledgments

This work grew out of a long series of discussions with David Barton, Richard Jenks and Barry Trager on how to organize the algebraic algorithms of MACSYMA. Without this nearly insurmountable problem, none of this work would have come to fruition. Other discussions with Jeff Arnold, Carl Hoffman and Jon Sieber have been invaluable. This work was supported by DARPA contract N00014-80-C-0622.

References

1. Daniel G. Bobrow and Mark J. Stefik, "LOOPS: An Object Oriented Programming System for Interlisp," *Proceedings of the European AI Conference* (1982).
2. Daniel H. H. Ingalls, "The Smalltalk-76 Programming System: Design and Implementation," *Proceedings of the Principles of Programming Languages Symposium* (1976).
3. Barbara Liskov, Alan Snyder, Russel Atkinson, and Craig Schaffert, "Abstraction Mechanisms in CLU," *Communications of the ACM* 20, (1977), 564-576.
4. Daniel L. Weinreb and David A. Moon, *Lisp Machine Manual,* MIT Artificial Intelligence Laboratory, Cambridge, MA, (1981).
5. Xerox Learning Research Group, "The Smalltalk-80 System," *Byte* 6, 8 (1981), 36-48.