CHAPTER II

Definitions and Framework

The context of the work presented here is established in this chapter by defining several terms used throughout this dissertation and discussing some relevant issues. The chapter begins with definitions for several common software engineering buzzwords in Section 2.1. The remaining sections address issues relevant to the design of reusable software, especially with respect to language design. Section 2.2 contains a discussion of the issues inherent to the design and implementation of reusable software components. Section 2.3 presents some characteristics of well-designed parts, and discusses their implications on programming language design. Section 2.4 contains a discussion of some issues relevant to the design of reusable components. Section 2.5 contains a discussion of several other issues relating to design of programming languages that support and encourage the design of reusable components. Section 2.6 concludes the chapter with a survey of existing languages evaluated with respect to the issues discussed in earlier sections.

2.1 Definitions

Several important terms have long been associated with software engineering. Essentially all software engineers use them, but unfortunately their definitions have not been standardized. The following are the definitions of these terms as used here.

Requirements are informal natural language statements of what a software component should do once it is designed and built. The design of a software component usually begins with requirements, which are often developed by (or by consultation with) a customer who needs the software but may or may not know precisely what he or she expects it to do. The job of a software engineer in doing "requirements analysis" (or more recently, "domain analysis") is to pin down exactly what the software needs to do.

Specifications are formal statements of what a software component does. A specification is a complete and unambiguous description of the component, and

constitutes a client's view of that component. Specifications are developed by the software engineer from the requirements and are the only official description of the component. In the event that requirements and specifications are inconsistent, specifications *always* indicate what happens. Specifications are formal statements, and in practice are mathematical assertions of some sort. For reasons discussed in Section 2.3.1, natural language statements, no matter how well-written, are not adequate for specifications.

Information hiding is a technique for keeping the irrelevant (e.g., implementation) details about a software component secret from all client programmers of that component[1]. There are two major reasons for doing this. First, if these details are revealed a client programmer may be confused by irrelevant information, making the component more difficult to understand. Second, once implementation details are revealed it is very tempting for a client programmer to take advantage of this information when using the component. This may make it difficult or impossible to upgrade the component later on with a new and improved version that has, say, better performance.

Abstraction is a technique for making a clear, comprehensible presentation of the information provided by a software component, explained in terms of "higher level" concepts than those used to implement the component. For example, abstraction is utilized when the operations provided by a character string manipulation package are defined using mathematical concepts such as "string" rather than implementation concepts such as bits, bytes, and null-terminated arrays of ASCII characters. Abstraction is important for the sole reason that without it, it would be all but impossible to reason about the behavior of any non-trivial software component.

It is important to realize that information hiding and abstraction are different and it is possible to have the former without the latter (but not vice versa). For example, consider a LIFO stack whose behavior is explained in terms of an array with a "top" index. If the client programmer is prohibited from directly accessing the representation of a stack, information hiding is enforced even though abstraction was not used to describe the component. Alternatively, a stack could be explained to the client in terms

---

1 Note that this definition hides information from client *programmers*, not just from client code. To many software engineers (e.g., [Meyer 88]), information hiding applies only to code. The client programmer may *see* internal structures and code, even though he or she may not write code that references them.

of a mathematical string (see Sections 2.4.2.3 and 3.1.1), and implemented using an array with a top index. If the implementation is not mentioned in the explanation of the behavior, both information hiding and abstraction are utilized.

2.2 Reusability Issues

The primary motivation for this research is the development of reusable software components. The concept of reusable software raises several legitimate questions. For example, what exactly is meant by the term "reusable software component?" What advantages are there to designing reusable components? If reusable components are so wonderful, why aren't they developed more often? What would characterize a good reusable component? Answers to these questions are presented in this section.

2.2.1 What Is A "Reusable Software Component?"

In this dissertation "reusable software" refers to software components that can be incorporated into a variety of programs *without modification* (except possibly parameterization). Furthermore, reusing these software parts should not compromise other software engineering principles such as information hiding and data abstraction. Like many popular buzzwords, reusable software has many conflicting definitions.

Some researchers (e.g., [Lanergan 84]) consider source code skeletons or templates to be examples of reusable software. In this approach, templates contain the structure of commonly used sections of code and/or declarations, which are essentially pasted into new applications and customized using an editor. This approach does not meet our definition of reusable software because the templates must be modified when they are reused, and information hiding and data abstraction principles are violated because the client has complete access to the source code.

A similar approach to software reuse is the development of new programs by modifying existing ones. This methodology is popular in the UNIX™ community because of the availability of source code for much of the system. This reuse of software does not meet the definition here because the component is modified, and information hiding and data abstraction principles are violated.

Transforming source code from one language to another (e.g., [Boyle 84]) is sometimes considered to be an application of reusable software because it permits a routine coded in one programming language (e.g., FORTRAN) to be "reused" within a program coded in another language (e.g., Ada). This approach fails our reusability test because the component is modified, and information hiding and data abstraction principles are violated.

Some researchers (e.g., [Jones 84]) consider every program that has been executed more than once to be reusable software, because every time the program is executed it is reused. This definition of reuse does not meet our definition because it is not concerned with reusable software components incorporated into client programs.

A simple example of a reusable software part is a routine from a run-time library where the object code for the routine is linked with a client's object code to form an executable image. It should, however, be possible to define reusable components that are more powerful and versatile than those found in traditional run-time libraries. So although the traditional run-time library does not completely demonstrate the power of reusable software, it does convey the gist of it.

An observation is in order at this point. It seems highly unlikely that a software component not intentionally designed to be reused would qualify as a reusable component. Thus, a corollary of this definition of reusable software is that reusable components are not written by accident or "scavenged" from existing code by "software archeology," but must instead be consciously and deliberately designed for reuse.

2.2.2 Advantages of Software Reuse

There are several reasons why designing and building reusable software components potentially improves both software quality and programmer productivity. First, because the cost of designing and building a component can be amortized over many uses, it is economically feasible to commit the time, intellectual energy, and money to do things right the first time. Committing necessary resources during the appropriate stages of the component's lifecycle improves quality in an obvious way.

Second, the designer of a reusable part knows that it will be used in applications unimaginable at the time of design, and will likely take the job seriously and design a

quality part. He or she will probably take the time to look at the larger picture, imagine and anticipate uses and variations of the part, and make the design general by factoring out idiosyncrasies of specific applications. In addition, the psychological effect of knowing one's design will be scrutinized by future programmers can have a positive impact on the quality of the part's design.

Third and perhaps most important, programmer productivity will increase because it is usually easier to reuse a well-designed software component than to design and implement one on the fly. This proposition is considered dubious by some who envision only very simple components as reusable, but for components with complex behavior (or especially those with simple behavior but complicated implementations) it is quite obvious. However, this claim does not seem to have been established by experimental evidence yet, due partly to the near-absence of truly reusable software components.

2.2.3 Non-Technical Impediments to Software Reuse

Why is reuse not more common? The answer involves a combination of technical and non-technical issues. The non-technical issues are discussed here, while the technical issues are presented in the next section.

One non-technical obstacle is an economic one. Software companies are in the business of producing and selling software. If a company sells a truly general and reusable component to a customer, that customer may no longer need the services of the software company. In order for reusable software to be economically profitable, a pricing structure and incentive system must be found that rewards manufacturers of reusable components without eliminating the economic market for the components.

Another obstacle is an organizational one. Manufacturers of reusable components must provide potential customers with detailed catalogs describing the available components, and customers must be able to efficiently search through these catalogs and easily determine whether a particular reusable component is appropriate for a particular application[2]. The customer should also be able to electronically order and immediately download selected components for incorporation into new projects.

---

2 Of course, these catalogs would not be the kind one reads in the library or uses to elevate toddlers at the dinner table, but instead would be an electronic database.

Negative psychological effects are another non-technical impediment to software reuse. The "Not Invented Here" syndrome is alive and well in the workplace, and programmers need encouragement to reuse someone else's component rather than develop their own home-brew version. Another form of this phenomenon might be called the "Macho Programmer Syndrome" where programmers (and indeed entire companies) regard software reuse as an admission that they are not as good as the other programmer, and therefore see reuse as a weakness. Of course, availability of reusable components that are significantly more reliable, efficient, and cost effective than custom-designed ones could provide the necessary incentives for programmers and companies to reuse software components.

Efforts are underway to address some of these obstacles (see, for example, [Biggerstaff 89a], [Biggerstaff 89b], [IEEE 87], and [IEEE 84]). However, the discussion in this dissertation does not address them.

2.2.4 Technical Impediments to Software Reuse

In addition to the non-technical obstacles just mentioned, there are several technical issues that currently keep reusable software from becoming a reality. In a very real sense solutions to these technical issues are more important than solutions to the non-technical ones, for until it is technically possible to design and build truly reusable components, management and organization cannot achieve widespread reuse of software.

The first technical obstacle is the lack of formal specifications for components. A programmer cannot be expected to reuse an existing part unless its functionality is crystal-clear. All too often programmers "reinvent the software wheel" because the functionality of existing parts is unclear or vague, and the alternatives — deciphering source code and trial-and-error testing — are often more painful than simply starting from scratch. A component will only be reused if its behavior is completely and unambiguously specified in a form understandable by potential programmers. As discussed in Section 2.3.1, these specifications should be mathematically rigorous. Specifically, informal natural language descriptions are not sufficient. Also, the principles of information hiding and abstraction should be followed, so providing the

client with source code of the component (as suggested in [Berard 89]) is not acceptable.

A second technical impediment is the inability to certify the correctness of a component. Of course, attempting to certify the correctness of a component whose specification is incomplete or ambiguous is an exercise in futility — how can one say whether a part behaves correctly when it is unclear exactly how that part is supposed to behave? Obviously the problem of formal specification must be solved before this issue can be meaningfully addressed. However, even with formal specification the problem of certification (i.e., formal verification) is difficult, in part because the techniques are still being developed and have generally not been applied to large programs with complex data structures. Also, many programming languages have constructs (such as aliasing) that significantly complicate program verification [Hoare 83, Krone 88]. Testing, a weaker certification method, has received much attention recently (e.g., [Mills 87]) , yet there are subtle barriers facing the would-be tester of a reusable software component [Hegazy 89].

Another technical obstacle is the relatively poor performance of reusable parts. Part of the problem here is the assumed trade-off between generality and performance that most programmers believe exists. In fact, there is no theoretical basis for this belief, although empirical evidence seems to support it. The problem is that most parts classified as reusable were designed and implemented using classical data structures and algorithms as taught in introductory computer science classes. These components, however, were not designed to be reusable, and performance suffers as a result. New evidence suggests that reusable parts can be designed that exhibit no significant performance degradation relative to non-reusable custom parts [Harms 89a].

2.3 Characteristics of Reusable Parts

Discussion in previous sections dealt with important reusability issues such as a definition for reusable software, why reusable software is a good thing, and some of the reasons why reusable software components are currently not designed and built very often. This discussion is concluded by presenting some characteristics that well-designed reusable components would exhibit. The discussion is on a general level and language-independent. However, the intent is to define some organizational structures

and constructs that should be included in any language supporting the development of reusable software.

2.3.1 Formal Specification

Every part must have a specification. A specification is essential for several reasons. First, it tells a potential client programmer exactly what the part does [Parnas 72]. This is important because a programmer cannot be expected to reuse a component whose function is ambiguous or vague. In addition, if information hiding is enforced (the desirability of which is assumed throughout this dissertation), a specification is the *only* way for a client programmer to know what a particular piece of software does.

Second, a specification tells the implementer of a part what the code must accomplish — abstractly, completely, precisely, and unambiguously [Parnas 72]. The implementer should not need any more information than what is presented in the specification. In particular, he or she should not need to know anything about the client, for if he or she did, the part would not be so reusable. Thus, a specification is the dividing line between a client and an implementer, constituting the only contract (and contact) between them.

Third, formal specifications are a necessity if formal verification is to succeed [Liskov 75]. An implementation of a part cannot be proved correct unless it is known precisely what the code is supposed to do. Of course, some researchers (e.g., [DeMillo 79]) consider formal verification as a hopelessly futile endeavor. However, others are much more optimistic about the prospects, making statements such as "Software engineering without mathematical verification is no more than a buzzword" [Mills 87]. One thing is for certain though — formal verification is not possible without formal specification.

A fourth and final advantage of formal specification is that the process of writing a formal specification often helps reveal ambiguities and inconsistencies in the part's design. This is because "formal specifications naturally lead the specifier to raise some questions that might have remained unasked, and thus unanswered, in an informal approach" [Meyer 85]. Thus, a formally specified part probably has a better design than one not subjected to the rigors of formal specification.

Obviously, the language in which specifications are written must permit (and preferably encourage) the expression of precise and unambiguous statements. Historically the languages best suited for this have been the languages of mathematics, such as first-order predicate calculus. In this view a specification consists of mathematical assertions of some sort (see Section 2.4.2).

But must specifications necessarily be mathematical in nature? Wouldn't it be just as good (or even better) for specifications to be well-written natural language statements rather than some mathematical mumbo-jumbo? It appears that the answers to these questions are "yes" and "no," respectively. A natural language is an inappropriate specification language for precisely the same reason it is such a wonderful medium for poetry and literature — it is easy to say one thing and mean another. [Meyer 85] provides a detailed example and analysis of the inherent dangers of English as a specification language.

Although natural language statements are not specifications, they can be instrumental in helping someone understand the function of a part [Parnas 72]. For this reason the designer of a part is encouraged to include an informal natural language description of the part with the part's definition. This description is included solely for use by programmers (both client and implementer) and is not intended for any kind of automatic processing (e.g., compilers and verifiers). Because a description is written in natural language it is by nature imprecise, and formally checking the consistency between it and the formal specification is impossible. In the event of a contradiction between the description and specification, the specification is *always* the final word.

Formal specification impacts the design of a programming language in several ways. First, mathematical assertions must be an integral part of the language, not simply considered as comments optionally entered by the programmer. Assertions should be required at strategic points in the code, such as pre- and post-conditions of operations and loop invariants. The language in which assertions are written must be powerful enough to completely specify a part's function, such as first-order predicate calculus. Specifically, propositional logic expressions are not sufficiently powerful. Of course, assertions may be ignored by the compiler, but it should be possible to mechanically process them by some other tool, such as a verifier.

Second, for verification to be possible, the semantics of control and data structuring mechanisms of the language must be be formally defined. Several methods are available to express the semantics of a language, including denotational semantics, axiomatic semantics, and operational semantics [Gordon 79, Marcotty 76]. Formal semantics influence language design because some programming language features are difficult to define formally and therefore should not be included in the language[3].

A consequence of this last point is that a language encouraging the development of formally-specified programs must be explicitly *designed* with this as a goal. It is very difficult to "retrofit" formal specification into an existing language, since it most likely has constructs not at all amenable to formal semantic description (and thus formal verification).

This is perhaps the primary flaw with specification languages that are defined independently of a programming language (e.g., Z [Spivey 89]). In these systems the formal specification of a software component is written in the specification language, and the actual implementation is coded in some programming language. This code can then be verified, assuming it has been decorated with appropriate assertions and the semantics of the implementation language are formally defined. This is unlikely unless the programming language was designed with formal specifications in mind, in which case the specification language would have been defined as part of the programming language, and would not be language-independent!

Very few programming languages offer a programmer the ability to formally specify a program. In fact, it is not possible to formally specify a program in any of the "popular" languages[4] (e.g., COBOL, FORTRAN, Pascal, C, and Ada). Also, some languages that appear to offer this capability don't actually have the power to write formal specifications (e.g., Eiffel, discussed in Section 2.6.7). Some languages that are exceptions to this are Alphard, Gypsy, and Euclid, which are discussed in Section 2.6.

---

3 Some language designers (e.g., [Wirth 83]) argue that features that are difficult to formally define would also be difficult for a programmer to understand, and thus should not be included in a language.

4 It is important to realize that what many programmers and languages call specifications (e.g., Modula-2's "definition module" and Ada's "package specification"˜) are merely signatures of exported items and not specifications in the sense used here.

2.3.2 Separation of Specification from Implementation

A second characteristic of a well-designed reusable component is the separation of the part's specification and implementation into separately-compilable units. This is beneficial for several reasons. First, it is perhaps the most effective method of realizing and enforcing information hiding. A client programmer is told what a part does by providing him or her with the part's specification. Placing the implementation someplace else and not allowing the programmer access to it prevents him or her from knowing anything about how that part is implemented. What better way to hide information!

Second, separating specification from implementation permits designers and client programmers alike to consider a part's abstract function and performance (e.g., space and time) as separate issues. This also opens up the possibility for a particular specification to have multiple implementations, each with different performance characteristics. Additional discussion about this is contained in Section 2.3.4.

Third, separating specification from implementation allows a client program using a part to be compiled and verified even before an implementation is written. (Of course an implementation must be written before the client can be linked and executed.) This makes it possible to have an implementer and client programmer coding independently, which is crucial in large programming projects. It also means that a client will not need to be recompiled (or reverified) when the implementation is selected (or changed).

This also implies that a client can be verified independently of any implementation. The verifier assumes that all parts referenced in the code are implemented correctly, which significantly reduces the amount of work the verifier needs to do [Krone 88].

Most modern languages supporting "modular" design (e.g., Ada and Modula-2) separate implementation from "headers" containing signatures of exported items. Since these headers do not contain formal specifications, not all advantages discussed here can be realized (e.g., verification). However, such languages have many of these advantages.

2.3.3 Generics

A well-designed reusable part should be generic if at all possible. A generic part is one whose definition is parameterized somehow (e.g., by a component type). For example, rather than defining separate parts for "stack of ints" and "stack of chars," it would be better to define a generic part for "stack of Item," where Item is a type parameter to the part. A generic part is not a usable part on its own, but is instead a *template* for a family of parts. When a template is instantiated a usable part is created that has arguments bound to all parameters. For example, instantiating "stack of Item" with type int creates a "stack of ints," and instantiating it with type char creates a "stack of chars." Generic parts achieve a degree of reusability in an obvious way.

Implementations of generic parts should not depend upon specific actual type parameters. For example, an implementation of stacks must implement stack of *any* component type, including stacks of ints and stacks of queues of chars. In other words, implementations of generic specifications must themselves be generic.

The ability to define generic parts is provided by many modern programming languages, including Ada, CLU, and Eiffel, which are discussed in Section 2.6.

2.3.4 Multiple Implementations

A particular specification has infinitely many possible implementations. Each one can be characterized by any number of performance characteristics, such as the amount of space required to store the structure, the amount of time necessary to access the structure, and the amount of space required for the implementation code. Many of these potential implementations exhibit such poor performance that no sane programmer would ever code them. However, most parts seem to have several implementations that exhibit "reasonable," yet different, performance. For example, one implementation of a part may require O(1) and O(n) time to insert and remove items from the structure, respectively, whereas another implementation may require O(log n) for both operations. Which of these two implementations is better? The answer is very dependent on the particular client, so it is not possible to say that one is universally better than the other.

A client programmer should be able to select an implementation from a list of available ones. This selection need not be made when the reusable part (i.e., the specification) is

chosen, but can instead be postponed until later, thereby factoring decisions regarding a part's function from its performance.

For flexibility, a client programmer should be allowed to choose an implementation for one part, and another implementation for a different instance of that same part (e.g., one stack implemented as a linked list, and another stack within the same client implemented as an array with top index[5]). Thus, implementation of a part is selected on an instance-by-instance basis, rather than on a client-by-client one.

Also, it should be possible for a client programmer to select a different implementation of a part without having to recompile or reverify the client. This allows a client's performance to be changed without materially altering its code, and allows a client programmer to take advantage of new implementations developed at a later time.

Multiple implementations affect the design of a programming language in several ways. First, multiple implementations of a part must be permitted, and the specification of a component must not be bound to any particular implementation or set of implementations. In other words, a specification is completely independent of its implementation(s), and should not be affected if and when a new implementation is developed[6].

Second, the mechanism used to bind a specification of a part to an instance of it within a client must be separate from the mechanism used to bind an implementation to that instance. In fact, it is conceivable at first glance that only the former binding (i.e., specification of an instance) be done in the client code, leaving the implementation binding to be specified at "link" time.

Third, the language must have constructs through which the performance characteristics of an implementation can be expressed in abstract terms (e.g., "big-O" notation). This is necessary because the client programmer must be able to understand the performance of an implementation without studying its code. (Besides, the client programmer doesn't even have access to the implementation code, since we're assuming information

---

5 Because of information hiding, the implementations should be described by abstract performance characteristics, not in terms of implementation structures such as linked lists and arrays.

6 The converse is not true — i.e., it is perfectly reasonable to bind an implementation to exactly one specification. This is because it is very unlikely that an implementation realizes more than one specification.

hiding.) One implication of this is that an implementation will not be described in terms of implementation structures (e.g., linked lists or arrays), but rather in abstract terms.

No modern programming language supports this criterion well. For example, Modula-2 binds every DEFINITION MODULE to exactly one IMPLEMENTATION MODULE. Ada effectively does the same, except that it's possible to write a package specification in such a way that multiple package bodies (i.e., implementations) can be defined. However, it is not possible to bind different package bodies to different instances of the same package within a single client.

Inheritance in object-oriented languages such as Eiffel is a mechanism whereby multiple implementations can be written for a class or operation. However, inheritance allows (and encourages) multiple implementations of individual operations within a class, rather than multiple implementations of entire classes. Section 2.4.1.1 contains a more detailed discussion of inheritance.

2.3.5 Efficient Implementations Possible

A part that has no efficient implementation will be neither used nor reused. This statement seems obvious, yet its impact is subtle, especially when considered in conjunction with the other characteristics discussed in this section. For example, many parts will be generic (e.g., stack of Item), and must be efficient for any component type. In other words, an implementation of stacks must be efficient for both "stack of ints" and "stack of queue of chars."

Perhaps the biggest implication of this is that unnecessary copies of data should not be made because copying is inefficient (usually taking time linear in the size of the representation data structure). This significantly impacts programming language design, especially with respect to parameter passing mechanisms. For example, Figure 1 shows an informal definition of a module providing a generic stack and operations Push, Pop, and IsEmpty[7]. This definition is in a hypothetical "Pascal-like" language, and is quite common in data structure texts, such as [Sedgewick 88].

---

[7] Note this is *not* the recommended definition of stacks. The definition that solves the problems inherent to this example is presented in Figure 2 in Section 3.1.1.

```
definition module Stack_Template (type Item);
  exports Stack, Push, Pop, IsEmpty;

  type Stack;

  procedure Push (var S:Stack; x:Item);
    -- places x onto stack S, without changing x

  procedure Pop (var S:Stack; var x:Item);
    -- requires stack S is not empty
    -- removes the topmost item from stack S, returning it in x

  function IsEmpty (S:Stack) : boolean;
    -- returns true iff stack S is empty

end Stack_Template;
```

Figure 1

Informal Description of a Template Providing Type Stack

One problem with this definition centers around the specification of procedure Push. Note that Push cannot change the actual parameter for x, effectively forcing a copy to be made of the item being pushed onto the stack. This is tolerable if type Item is a simple type, such as int or char, but it must be considered unacceptable in general, especially if there is a more efficient alternative.

The alternative suggested by most programming languages (and programmers) is to pass a pointer to the item being pushed, rather than passing a copy of the item. This could be accomplished either explicitly (e.g., make the type of parameter x "pointer to Item") or implicitly (e.g., make x a "call-by-reference" parameter). This solves the problem at hand. However, introducing pointers at this level makes formal specification and verification difficult (see Section 2.3.1) and has other significant problems discussed in Section 2.5.1. Pointers are not the answer.

If copying data structures is bad, and introducing pointers into the language is bad, what other options are there? To date, no language has offered any other alternative. However, in Section 3.3.1, swapping the values of two variables is introduced as a data movement primitive that solves the problem of moving large data structures efficiently and does not introduce pointers into the language definition.

As just mentioned, no modern programming language offers anything other than pointers as an alternative to copying large structures. Consequently, no modern

language permits implementations of generic parts to be both formally specified and efficient. This is strong evidence supporting the thesis that no modern programming language has all of the constructs necessary to encourage and facilitate the design of reusable software parts. In fact, existing languages have constructs, such as implicit aliasing, that actually thwart design of and with reusable software components.

2.4 Reusable Part Design Issues

The discussion in the previous section dealt with characteristics of reusable parts. However, there are at least two open issues regarding the design of reusable components:

- What functionality should be incorporated into a reusable part?
- How does one go about formally specifying a reusable part?

These issues are addressed in the following subsections.

2.4.1 Encapsulation

A reusable part is usually defined by a module encapsulating a number of different kinds of items — e.g., types, operations (i.e., procedures and/or functions), state variables, and constants — which are provided to a client of the part. One of the tasks of a designer of a part is to decide which items are provided by the reusable part. These decisions strongly influence the "flavor" of the part as well as the extent to which the part may actually be reused.

In this section two approaches to module design are discussed — the Abstract Data Object approach and the Abstract Data Type approach. Throughout this discussion it is important to understand that these approaches are simply two reference points on the "modularization approach continuum." The design of most modules actually falls at some intermediate point on this continuum. However, the classification is useful, since programming languages generally support and encourage module design skewed toward one of these reference points.

Section 2.4.1.3 concludes with a brief discussion of an important classification of operations — primary vs. secondary.

2.4.1.1 Abstract Data Objects

In the Abstract Data Object (ADO) approach to encapsulation, data and operations to manipulate that data are encapsulated in something called an object. A module defines a template for a class of objects, and a client uses templates to declare objects. The data portion of an object is not directly available to the client. Rather, the only way for a client to do anything with the data is by invoking its attached operations.

For example, the following is an informal definition in a hypothetical language of a module defining a template for stacks of integers, where the available operations on a stack are Push, Pop, and IsEmpty[8].

```
class Stack is

  procedure Push (x:integer);
    -- places x onto the stack without changing x

  function Pop () returns integer;
    -- requires stack is not empty
    -- removes the topmost item from the stack, returning it

  function IsEmpty () returns boolean;
    -- returns true iff the stack is empty.

end Stack;
```

A client would use the variable declaration facility to declare Stack objects. For example:

```
var s1 : Stack;
var s2 : Stack;
```

declares two Stack objects, called s1 and s2. Conceptually, each Stack object (i.e., variable) contains the representation for a stack as well as code for operations Push, Pop, and IsEmpty. Pushing the value of integer variable y onto stack s1 would be accomplished by the statement:

```
s1.Push (y);
```

---

8 ADO modules do not need to define data because a client cannot access it. The only reason for having data declarations would be to specify the effect of the operations. Since this is just an informal description data declarations are not included.

Note the use of the "." in this statement. The object being manipulated is s1 and the operation being performed is the Push operation — not just any Push, but the one belonging to object s1.

In the ADO approach, an object is treated as a black box with one or more buttons on it. Each button (i.e., operation) may have slots around it (i.e., the operation's parameters) into which items may be placed, and from which items may appear. The meaning of each button is described in some sort of owner's manual (i.e., specification). To use the box, necessary items are placed in the appropriate slots, a button is pressed, the internal mechanisms of the box churn, and any items produced by the operation are removed from their slots.

This analogy suggests other important characteristics of the ADO approach. For instance, the only purpose of an operation (i.e., button) is to do something to its associated object (i.e., black box). In other words, *every* operation must be explicitly attached to exactly one object. An operation affecting more than one object (e.g., procedure Push in the above example) is attached to one of the objects (e.g., a Stack), and the rest are passed as parameters (e.g., integer x). This asymmetry is quite natural for many operations, such as those for Stacks, but makes the definition of others seem awkward. For example, a function that returns the sum of two integers is conceptually symmetric, yet in the ADO approach it must be attached to one of the integers. The function invocations "`x.add(y)`" or "`y.add(x)`" are awkward ways of calculating the sum of x and y, which is expressed as "`x + y`" or "`add(x,y)`" in traditional languages.

As a second example of ADO characteristics, suppose a client wished to define an operation that reverses the items on a Stack[9]. This operation must be attached to an object, and it seems reasonable to attach it to Stacks. Because of information hiding, however, the client cannot modify an existing class. Instead, the client might define a new class that *inherits* Stacks and defines an additional operation. The new class is often called an heir or descendant of the inherited (or ancestor) class. For example, the following might be the syntax (again in the hypothetical language):

```
class Rev_Stack inherits Stack is
```

9 This can be accomplished by invoking operations Push, Pop, and IsEmpty, and using a temporary FIFO queue. In other words, access to the internal representation of Stacks is not necessary.

```
    procedure Reverse ();
      -- Reverses the stack

end Rev_Stack;
```

Rev_Stack objects have operations Reverse, Push, Pop, and IsEmpty. Also, since Rev_Stack objects inherit all operations of Stacks, they can be used in any context where a Stack object is legitimate. For example, if a parameter to an operation needed to be a Stack, a Rev_Stack would work just as well.

Inheritance can also be used to "redefine" an inherited operation, allowing (presumably minor) modification of the behavior of the inherited class. Assuming that information hiding is observed, these modifications must be coded in terms of operations defined in the inherited class[10]. The specification of the operation should also change to reflect the modification, which may have repercussions on other operations. For example, class Rev_Stack could redefine procedure Push. If an operation that needed a Stack object was given a Rev_Stack instead, all invocations of this parameter's Push invoke the procedure redefined in Rev_Stack rather than the one defined in Stack. Thus, when the operation invokes Push, it has no way of knowing which Push will actually be invoked. The ability to modify an operation increases the power and flexibility of a class, but complicates formal specification and verification, for obvious reasons.

The ADO approach underlies what is commonly called "object-oriented programming," which is encouraged and facilitated by programming languages such as Simula-67, C++, CLU, Smalltalk-80, and Eiffel. Note, however, that our definition of the ADO approach factors out several characteristics sometimes associated with object-oriented languages, such as polymorphism, dynamic typing, "typeless" variables, and variables as object references. These are considered to be characteristics of specific languages rather than characteristics of an encapsulation approach. Consequently, they are discussed in the context of specific languages in Section 2.6.

---

10 Allowing a descendant to have direct access to its ancestor's representation increases the usefulness of inheritance, and many "object-oriented" languages allow it, even though it directly violates the principle of information hiding.

2.4.1.2 Abstract Data Types

In the Abstract Data Type (ADT) approach to encapsulation, a module (sometimes called a package) defines one or more types and operations involving parameters of those types. A client declares variables of the provided type, and a variable is considered to contain a value from the domain of its type. A client cannot directly access the internal representation of the variables, but must pass them as actual parameters to operations.

For example, the following is an informal definition in a hypothetical language of a module providing a type for stacks of integers and operations Push, Pop, and IsEmpty:

```
module Stack_Module is

  type Stack;
    -- type for stacks of integers.

  procedure Push (s:Stack; x:integer);
    -- places integer x onto stack s, without changing x.

  function Pop (s:Stack) returns integer;
    -- requires stack s is not empty
    -- removes the topmost item from stack s, returning it

  function IsEmpty (s:Stack) returns boolean;
    -- returns true iff stack s is empty.

end Stack_Module;
```

A client would use the variable declaration facility to declare stacks. For example,

```
var s1 : Stack;
var s2 : Stack;
```

declares two stacks variables, called s1 and s2. Pushing the value of integer variable y onto stack s1 would be accomplished by the statement:

```
Push(s1,y);
```

and pushing integer z onto stack s2 would be accomplished by:

```
Push(s2,z);
```

Note that conceptually there is now only one Push operation, which pushes an item onto any stack it is given.

Operations can be defined independently of a type, in the sense that they can be declared outside of any module providing a type. For example, suppose a client wanted to declare a procedure to reverse the items on a Stack (which does not need access to the representation of a Stack). He or she could easily write a procedure called Reverse that has one parameter of type Stack. This procedure could be invoked with any Stack variable.

In a "black-box" analogy similar to that presented for the ADO approach, an ADT variable is treated as a black box containing some data but no operations. Each operation is treated as a machine. Each machine has exactly one button, and may have one or more slots (i.e., the operation's parameters) into which data may be placed, and from which data may appear. The meaning of each machine is described in an owner's manual (i.e., the operation's specification). To use a machine, necessary data items are placed in the appropriate slots, the button is pushed, the machine munges with the data items, and any data items produced are removed from their slots.

Some popular languages that encourage and facilitate this style of programming are Modula-2 and Ada. Specific features of these languages are discussed in Section 2.6.

2.4.1.3 Primary and Secondary Operations

When designing a module, there are potentially many operations that could be included in it. For example, reversing a stack is an operation that is occasionally useful. Should it be included as an integral part of stacks, or relegated to the status of an operation that can be added later? A classification of operations that is useful in this context distinguishes between *primary* operations and *secondary* ones.

Primary operations are those that are necessary to define the "essence" of the structure, in the sense that all "useful" operations can be defined in terms of them. All other operations are secondary ones, and can be implemented by invoking combinations of primary operations[11]. For example, operations Push, Pop, and IsEmpty are primary operations for a stack, since they are essential in defining a LIFO structure. On the other hand, Reverse and Copy are secondary operations because they can be coded by invoking the primary operations.

---

[11] This distinction is similar to that made in [Parnas 72] between "normal" functions and "mapping" functions.

2.4.2 Specification

Section 2.3.1 argues that formal specification is a necessary characteristic of reusable software parts, though it does not address how one goes about formally specifying a part. This section presents some background for formal specification, and discusses two popular approaches to constructing formal specifications — the algebraic approach, and the model-based approach[12].

2.4.2.1 Mathematical Theories

Formal specification, as argued in Section 2.3.1, must be mathematical. Therefore, let's take a closer look at what mathematical theories are, and how they are developed.

A typical mathematical theory is characterized by a *signature* and a set of *axioms*. The signature is a set of symbols that are defined by the theory. These symbols stand for mathematical types, mathematical constants, or mathematical functions. The axioms are statements that define the mathematical type by defining its domain and a notation for expressing functions and relations among the values in this domain.

Proofs in a theory construct statements (or formulas) from the axioms using certain rules of inference, such as modus ponens and substitution. These formulas are the *theorems* of the theory. Conversely, a formula is said to be provable (i.e., a theorem) if and only if there exists a proof for constructing it. In addition, *definitions* are often made, which are simply short-hand notations for use in writing formulas.

For example, one characterization of mathematical string theory over an alphabet Σ has the signature (Σ^*, Λ, C) and three axioms. Here, Σ^* is the mathematical type of strings over Σ, Λ:Σ^* is a particular (constant) string, and C:$\Sigma^* \times \Sigma \rightarrow \Sigma^*$ is a function. These symbols are implicitly defined by the following axioms:

I. $\forall \alpha$:Σ^*, $\forall$z:Σ,
C(α,z) $\neq \Lambda$

II. $\forall \alpha,\beta$:Σ^*, $\forall$y,z:Σ,
(C(α,y) = C(β,z)) $\Rightarrow$ (α = β $\wedge$ y = z)

---

12 Other specification techniques, such as state machines and axiomatic descriptions, are discussed in [Liskov 75].

III. $\forall S \underline{1} \Sigma^*, \forall \alpha{:}\Sigma^*, \forall z{:}\Sigma,$

$((\Lambda \in S) \wedge (\alpha \in S \Rightarrow C(\alpha, z) \in S)) \Rightarrow S = \Sigma^*$

Some useful definitions might be:

$Cat : \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$

(i) $Cat(\alpha,\Lambda) = \alpha$

(ii) $Cat(\alpha, C(\beta, z)) = C(Cat(\alpha,\beta), z)$

$Len : \Sigma^* \rightarrow N$

(i) $Len(\Lambda) = 0$

(ii) $Len(C(\beta, x)) = Len(\beta) + 1$

Theorems of string theory are constructed from these axioms and definitions using the rules of inference. For example, the following is a theorem:

$\forall \alpha,\beta{:}\Sigma^*, Len(Cat(\alpha,\beta)) = Len(\alpha) + Len(\beta)$

For a theory to be useful, several properties must be proved. One of these is *consistency*, which says that it is impossible to prove both a formula and its negation (e.g., it is not possible to prove both "$\forall \alpha,\beta{:}\Sigma^*, Len(Cat(\alpha,\beta)) = Len(\alpha) + Len(\beta)$" and "$\forall \alpha,\beta{:}\Sigma^*, Len(Cat(\alpha,\beta)) \neq Len(\alpha) + Len(\beta)$").

Theorems and axioms are just sequences of symbols with no inherent meaning or notion of truth. In other words, theorems have syntactic structure but no semantic meaning. Of course, the most important function of mathematics is its ability to say things *about* something, so a theory is practically useless unless its theorems and axioms have meaning.

Meaning is associated with a theory by modeling (or interpreting) each symbol by some meaningful entity. For example, one meaningful model of string theory interprets the constant Λ as the empty string, and the value of the function $C(\alpha, z)$ as the string obtained by attaching the symbol z to the right-hand end of the string α. The definition Cat corresponds to concatenating two strings together, and definition Len is the length of a string. This interpretation associates semantic meaning to every axiom and theorem of string theory. For example, the theorem mentioned above means that the length of the concatenation of two strings is the sum of the lengths of the two strings. Although

this is the intended and most straightforward model for string theory, there are many other ones.

There are many statements in the vocabulary of string theory whose meaning under a particular interpretation is either invalid or nonsensical. For example, under the interpretation for string theory given in the previous paragraph, the statement "ΛΛ`Cat(Λ,Λ)Len(x)`" is nonsensical, while the statement "∀α,β:∑*, `Len(`α`) = Len(`β`)`" is invalid. If either of these statements happened to be theorems in string theory, we'd be in big trouble!

In fact, for an interpretation to be a useful model of a theory, two important properties must hold between the theory and the interpretation — soundness and completeness. A theory is *sound* with respect to an interpretation if and only if the meaning of every theorem under the interpretation is valid. For example, if string theory is sound with respect to our interpretation, the meaning of every theorem of string theory is valid. In other words, no theorem would have an invalid or nonsensical meaning. Soundness is crucial for obvious reasons.

A theory is *complete* with respect to an interpretation if and only if every valid statement under the interpretation is a theorem of the theory. Completeness says that it is possible to construct a theorem for every statement that is valid. For example, in our interpretation of string theory the statement "`Cat(Cat(`α`,`β`),`γ`) = Cat(`α`,Cat(`β`,`γ`))`" is valid. If string theory is complete with respect to this interpretation, this statement is also a theorem.

Showing that a theory is both complete and sound with respect to an interpretation is a non-trivial task, and requires a formal semantic definition of the interpretation. Without these results, however, a theory is practically useless.

2.4.2.2 Algebraic Specification

In the algebraic approach to formal specification, a mathematical theory is developed for each data structure. For example, Guttag [Guttag 77] develops the following theory for "Stacks of T" (denoted S(T)):

`Signature is (S(T),` Γ`, Push, Pop, Top, IsEmpty), where`

Γ : S(T)

Push : S(T) × T → S(T)

Pop : S(T) → S(T) ∪ {⊥}

Top : S(T) → T ∪ {⊥}

IsEmpty : S(T) → B

Axiom Schemas are

I. IsEmpty(Γ) = true

II. IsEmpty(Push(s,x)) = false

III. Pop(Γ) = ⊥

IV. Pop(Push(s,x)) = s

V. Top(Γ) = ⊥

VI. Top(Push(s,x)) = x

In the intended interpretation of this theory, constant Γ is the empty stack, function IsEmpty(s) is true iff s is the empty stack, Pop(s) returns the stack obtained by removing the top element from s, and Top(s) returns the top element of s without removing it. Also, it is an error to attempt to Top or Pop from the empty stack.

Several important questions must be answered if we are to have any confidence in a theory developed in this manner — is the theory consistent, is the theory sound with respect to our intended interpretation, and is the theory complete with respect to this interpretation? In other words, does the theory (e.g., Stack Theory) *exactly* capture our intuitive notion of the data structure (e.g., LIFO stacks)? As mentioned in the previous section, proving soundness and completeness are non-trivial exercises. However, they must be done for each theory developed.

Also, since these theories are "brand new" there are no theorems available to help a verifier prove the correctness of a program. Developing a set of useful theorems is also non-trivial.

Another potential problem with this approach is visualization — specifically, axioms are not easy to visualize. For example, a stack is usually visualized by drawing pictures or having in mind some physical model (e.g., a "stack of cafeteria trays"). In these visualizations notions like "top" and "empty" make a lot of sense. On the other hand, no images pop into mind (pun intended!) with the axiom "Pop(Push(s,x)) = s".

However, the lack of visualization may simply be a problem with our current thinking, and given enough practice with these theories, visualizations may appear quite naturally.

Assuming that a theory is sound and complete, how does one use it to specify a reusable part? There are two approaches to this. In the first approach, all operations provided by the reusable component are defined as mathematical functions in the theory. In a sense, the theory is not merely *used* to specify the part, it *is* the specification. One problem with this is that adding a new operation to a part involves rewriting the theory and consequently, reestablishing soundness and completeness. It also means that theories may have more functions defined than absolutely necessary. For example, a theory for stacks can be defined without function IsEmpty or axioms I and II [Liskov 75]. However, if the reusable part provides an IsEmpty operation, IsEmpty and the corresponding axioms must appear in the theory. This approach is promoted in [Guttag 78] and [Musser 80].

A second approach to using these theories is to model the data provided by the reusable part as a value from the domain of the mathematical type. Operations are defined with assertions (i.e., pre- and post-conditions) involving functions and relations defined in the theory. This is actually a variant of the model-based approach discussed in the next section, and more will be said there. This approach is encouraged by languages such as Larch [Wing 87, Guttag 85].

2.4.2.3 Model-Based Specification

Unlike the algebraic approach to specification, the model-based approach specifies reusable parts in terms of existing mathematical theories. The data provided by a part is modeled as a value from some mathematical domain. For purposes of reasoning about program data, a corresponding mathematical value is considered to be its mathematical model. Operations are defined with pre- and post-conditions, which are assertions about the values of the operation's parameters before and after the operation invocation. Within these assertions parameters are considered to be mathematical values of the appropriate type, and the assertions involve functions and relations in the theories defining those types.

For example, a "stack of T" can be modeled as a "mathematical string of T," with the top of the stack at the right-hand end of the string. For purposes of reasoning, stacks are

considered to be mathematical strings. The effect of operations Push, Pop, and IsEmpty can be expressed formally as assertions in string theory:

```
procedure Push(s:Stack,x:T)
  ensures "s = C(#s,x) and x = #x"

procedure Pop(s:Stack,x:T)
  requires "s ≠ Λ"
  ensures "#s = C(s,x)"

function IsEmpty(s:Stack) : boolean
  ensures IsEmpty iff "s = Λ"
```

In this notation, a requires clause specifies the pre-condition that is expected to be true when the operation is invoked, and an ensures clause specifies the post-condition that is guaranteed to be true when the operation returns. Within an ensures clause, a '#' in front of a parameter denotes the value of that parameter at the beginning of the operation. For instance, the effect of Push(s,x) is that upon completion, the value of s (treated as a string over alphabet T) equals the old value of s with x appended to the right-hand end, and x is unchanged.

A major advantage to the model-based approach is that existing mathematical theories are "reused." This means that the specifier can take advantage of the knowledge gained over the years by mathematicians — for example, completeness and soundness properties of the theories, and a wealth of theorems.

Some might argue, though, that many interesting data types cannot be appropriately modeled by "common" mathematical theories such as number theory and string theory. Recent research, however, has demonstrated that a large number of useful reusable parts can be appropriately (and intuitively) specified in terms of surprisingly few mathematical theories. These theories include function and relation theory, set theory, number theory, string theory, and graph theory [Weide 86a].

Another advantage is that these theories are easy to visualize, and drawing pictures is a good way to understand a part's formal specification.

2.5 Programming Language Design Issues

Section 2.3 presents characteristics of well-defined reusable parts, and discusses the impact these characteristics have on programming language design. This section "inverts" the presentation by discussing programming language issues and their impact on the design of reusable parts. Included in the discussion of each issue is the approach taken in RESOLVE.

The issues discussed in this section relate to pointers (Section 2.5.1), data movement and parameter passing (Section 2.5.2), and types and variables (Section 2.5.3).

2.5.1 Pointers

Pointers (a.k.a. references and indirect addresses) are one of the most fundamental and powerful concepts in modern computer architectures, which allows a value to be interpreted either as a value or as the address of a value. Fundamental machine structures such as program counters, stack pointers, base registers, and return addresses are based on this idea. In fact, it is practically impossible to imagine a machine language program that does not rely on pointers. However, pointers are also one of the most dangerous aspects of machine code programming because of the potential for confusion between an address and its contents. Program bugs resulting from misuse of pointers are notoriously disastrous and difficult to find and correct.

A good high level language protects the programmer from the ugliness and dangerousness of the actual machine while enhancing the programmer's ability to solve problems. Unfortunately, high level languages have not been successful at hiding all details of the machine from the programmer. For example, pointers in one form or another appear in just about every language. In some languages, such as FORTRAN, pointers are implicitly part of the language definition because they are necessary to explain parameter passing to subroutines (which is discussed in more detail in the next section). Pointers are also implicit in many "object-oriented" languages, such as CLU [Liskov 81] and Eiffel [Meyer 88], because all class variables are defined as references to the actual class structure. Knowing these implicit uses of pointers is essential to understanding the meanings of many constructs in these languages.

Not only are these implicit pointers dangerous, the potential for implicit aliasing tends to thwart formal specification and verification efforts. Consequently, languages that include them may be unsuitable for creating reusable parts. The problem essentially boils down to the following. If a variable is not mentioned in a statement, it is usually assumed that its value does not change during execution of that statement. However, if uncontrolled aliasing is allowed, the semantic definition of every statement must take into account the possibility that any variable — even one not mentioned in the statement — may have its value changed as a side-effect of executing that statement. Incorporating this into the formal semantic definition of a language greatly increases the complexity of the definition, and therefore the complexity of reasoning about program behavior. For example, much of the complexity of the proof rules for Pascal found in [Hoare 73] and [Luckham 79] is a consequence of implicit aliasing.

In Pascal and Pascal-like languages, such as Modula-2 [Wirth 82] and Ada [DoD 83], pointers are also apparent since the programmer is allowed to explicitly declare pointer variables. Programmers in these languages not only have the power of machine addresses available to them, they also have the potential problems that come with the territory, such as unintended aliasing and dangling references.

The appropriateness of including pointers in programming languages has received considerable attention within the computer science community. C.A.R. Hoare, for example, feels “their introduction into high-level languages has been a step backward from which we may never recover” [Hoare 83]. The problem is not so much with formally specifying pointers, since their specification is very similar to the specification for arrays [Luckham 79]. Rather, the problem is the temptation to use pointers inappropriately, unnecessarily increasing the complexity of the specification and proof. For example, consider the following similar Pascal segments (which are provable using [Hoare 73] and [Luckham 79]):

```
var x,y : integer;
...
{assert true}
x := 3;
y := 5;
{assert x=3}
```

```
var x,y : ^integer;
...
{assert x≠nil and y≠nil and x≠y}
x^ := 3;
y^ := 5;
{assert x^=3}
```

Note that the segment using pointers is more complicated to specify and verify than the one without pointers because a precondition concerning the values of the pointers must

be specified and proved. In fact, note that the precondition involves variables not even mentioned in the postcondition (i.e., y^). In general, the precondition involves all variables of pointers to types mentioned in the postcondition (e.g., ^integer in the above example).

Some language designers (e.g., [Meyer 88]) argue that pointers should be defined in programming languages because they are necessary to efficiently implement traditionally "linked" structures, such as lists. Our research, however, has shown this argument to be invalid — indeed, it is possible to formally define modules that safely encapsulate the notion of "pointer" for use in implementing linked structures. One of these modules is discussed in Section 3.6.2, and others can be found in [Pittel 90] and [Weide 86b]. If these modules are definable by and available to the programmer, there is no need to include pointers as a part of a programming language, which is the approach taken in RESOLVE.

2.5.2 Data Movement Primitives

An essential feature of any programming language is the ability to move data from one place to another. Modern languages provide only one mechanism to do this — copying the data. Examples of the (implicit) use of copying are the assignment statement and call-by-value and call-by-value-result parameter passing mechanisms.

For example, consider the assignment statement "x := y" where x and y are variables of the same type. The usual semantics of this statement is that, after it is executed, the values of variables x and y are the same, and furthermore, that this value is equal to the value of y immediately before the assignment statement. In implementation, this means making a copy of the value in y and placing it in variable x. Parameter passing by value and value-result imply copying in the same sense.

Although the ability to make a copy of a variable's value is occasionally necessary, reliance on it to the point that copying is implicit — and in some languages even essential in order to program at all — is unwarranted and counterproductive to designing reusable parts. Specifically, three problems are inherent to copying:

- Physically copying a large data structure is usually inefficient.

- Copying a pointer is not tantamount to copying the data structure it points to, and often produces programs that are incorrect and difficult to debug.

- "Copy thinking" leads to design of generic procedures having *no* efficient implementation.

As mentioned in Section 2.3.5, the time required to physically copy a data structure is usually linear in the size of the structure. Copying simple values, such as integers and characters, is relatively cheap, whereas copying large structures, such as records or linked structures, is potentially quite expensive, resulting in intolerable performance.

Most current languages attempt to overcome this inefficiency by copying pointers to data structures rather than physically copying the structures themselves. These pointers may appear in the language explicitly through a pointer type, or implicitly through call-by-reference parameter passing. As mentioned in Section 2.5.1, the introduction of pointers into a language causes potential problems with reasoning about programs and interferes with program verification.

In addition, pointers conflict with "hidden" types definable in many languages (e.g., private types in Ada and opaque types in Modula-2). For example, consider the assignment statement "x := y" where the representations of variables x and y are hidden. This statement either makes a copy of the data structure contained in y (if y is the actual data structure) or it makes a copy of a pointer to the data structure (if y is a pointer to the data structure). Unfortunately, the client programmer has no way of knowing which of these two semantically inequivalent actions is taken, since the representation is hidden[13].

There is at least one more problem with "copy thinking." Because copying is the only built-in data movement primitive defined in current programming languages, an entire generation of reusable software components has been designed in such a way that they cannot possibly be implemented efficiently. An example of such a component was presented in Figure 1 and discussed in Section 2.3.5.

Two key notions regarding parameter passing are apparent from the above discussion. First, passing parameters is a primary means of moving data in a program. Second,

---

13 Some (e.g., [Meyer 88]) would argue that this situation justifies allowing the client programmer access to the implementation. However, the problem appears to lie not with information hiding or hidden types, but with pointers.

every parameter passing mechanism in existing languages (e.g., call-by-value, call-by-value-result, call-by-reference[14]) either copies data structures or is defined in terms of pointers to them (with the potential for implicit aliasing). Either way, none of these parameter passing mechanisms successfully addresses the issues raised in this discussion.

An alternative to copying is swapping the values of two variables. Swapping as a data movement primitive successfully addresses all three problems inherent to copying — it can be implemented as a constant time operation (i.e., it is efficient for all types), it is defined without the need to mention pointers (and it does not have problems of implicit aliasing), and it suggests a "swapping-style" of generic procedure design that can be efficiently implemented. In RESOLVE, swapping is the only built-in data movement primitive. One implication of this is that all parameters are passed "by-swapping." A complete discussion of the advantages and consequences of this is presented in Section 3.3.1, and also in [Harms 88] and [Harms 89b].

2.5.3 Types and Variables

The concept of data type is incorporated in some form or another into every modern imperative programming language, and is usually coupled with the notion of variables. This section explores these two ideas — types and variables — and their relationship to programming language design.

We'll start by examining several common views of types and their role in programs and programming languages (Section 2.5.3.1 and 2.5.3.2). Sections 2.5.3.3 through 2.5.3.6 address the issues of type equivalence, type coercion, dynamic vs. static typing, and type initialization/finalization, respectively. This section concludes with a discussion of built-in data types, presented in Section 2.5.3.7.

2.5.3.1 Meaning of Types and Variables

There is very little concensus in the computer science community about the meaning of "data type" and "variable," although practically every programming language is defined in these terms. About the only thing all definitions of data type have in common is the

---

[14] Definitions of these and other parameter passing mechanisms can be found in most programming languages texts (e.g., [Pratt 84]).

notion that a type is, at least partially, a set of values. There is little agreement on what these values are, or on what else is encapsulated with this set.

In most languages designed prior to the mid-1970's (e.g., Pascal [Wirth 74]) a data type is simply a set of values a variable may assume. These languages provide a number of primitive built-in types (e.g., integers, characters, and booleans), and constructors for creating aggregate types out of simpler ones (e.g., arrays and records). All types are defined in terms of these primitive types or constructors. This scheme has the advantages of simplicity and intuition, and seems to follow nicely from the notion of types from mathematics.

However, it also has two significant drawbacks — it incorporates neither abstraction nor information hiding. Abstraction cannot be used in defining a type because all types are defined in terms of implementation-level building blocks. Information hiding is not achieved because a programmer knows about the implementation of every type and can write code that takes advantage of this information. For example, type stack must be defined in implementation terms such as record, array, and integer. The fact that there are operations such as push and pop has nothing to do with the definition of type stack. Worse yet, a programmer can write code that directly accesses the fields of the stack record.

A popular approach that addresses information hiding defines a data type (sometimes called an abstract data type) to be a set of values together with the set of allowable operations on those values, similar to the *class* concept in Simula-67 [Dahl 70]. Ada , CLU, and Eiffel are some of the languages that adhere to this definition, and in fact, this is the definition presented most often in the literature (e.g., [Wing 87], [Guttag 78]) and in recent texts on programming languages (e.g., [Pratt 84], [Horowitz 84]).

With this approach, it is possible to define a module (variously called a package, class, or cluster) that encapsulates a set of values and a set of operations on those values. The actual implementations of the set of values and the operations are hidden from a client of the module, so the only means of accessing the values are by invoking the associated operations. For example, accessing a stack must be done by invoking operations such as push and pop. The obvious advantage to this is that information hiding is enforced.

There are, however, several problems with this approach. First, associating operations directly with a type's definition does not match the intuitive notion of type[15]. Types are usually discussed within the context of data, which is intuitively a passive quantity that is acted upon by operations, which are active agents. Defining a data type (intuitively passive) as a set of operations (intuitively active) seems quite strange[16]. Even proponents of this approach seem confused about this. For example, the Ada reference manual talks about a "value of type integer" [DoD 83], which doesn't make any sense if type integer is considered to be a set of values together with a set of operations. On the other hand, this phrase makes a lot of sense if type integer is considered to simply be a set of values. Also, in many of these languages (e.g., Ada) an identifier defined as a "type" essentially represents only a set of values, and does not have any operations associated with it, which is not consistent with other uses of the term type.

A second problem with this approach deals with operation extensions. Taken literally (as is done in languages such as CLU), the only operations available for a type are those that define the type. For example, if type stack were defined with operations Push, Pop, and IsEmpty, it would not be possible to later define an operation Top that returns the top item from a stack without popping it. This is somewhat anti-convenient, especially since this particular operation can be implemented independently of any realization of stacks by invoking operations Pop and Push. Thus, the designer of a module must anticipate all possible operations that would ever be useful for that type, which is neither possible nor desirable. This limits the "reusefulness" of the type.

A third and final problem here is that the issue of abstraction is not addressed by this approach. As defined in Section 2.1, abstraction is a technique for making a clear, comprehensible presentation of the information provided by a software component, explained in terms of "higher level" concepts than those used to implement the component. A data type in this approach is obviously not defined in terms of

---

15 Some would argue that this intuition is wrong. However, a definition of types that is both intuitive and addresses the concerns raised in this discussion would be better than one that addresses the concerns but is unintuitive. RESOLVE's definition of types (discussed shortly) falls into the former category.

16 An interesting approach is defined in Russell [Donahue 85]. In Russell, a data type is a set of operations that provide meaning for untyped values from a universal value space. (An analogy is drawn to computer hardware, where the untyped universal value space is the set of all possible bit strings storable in a computer's memory. Each of these strings is inherently untyped. Typing is provided by instructions (i.e. operations) operating on the value.) In other words, a data type does not consist of a set of values, but only a set of operations.

implementation-level concepts, which is good. Unfortunately, it is not defined in terms of *any* concepts, either high-level or low-level! In fact, data types defined this way are no more abstract than comparable ones defined in earlier languages such as Pascal, and the term "abstract data type" is actually inappropriate in this context[17].

What is needed are mechanisms that hide information from clients (i.e., implementation detail) while at the same time provide necessary information to clients (i.e., abstract specifications). RESOLVE accomplishes these goals by separating types and variables into two groups — mathematical and program.

A mathematical type is the name given to a mathematical domain (i.e., a set of abstract values defined by a set of axioms). For example, mathematical type "integer" might be the name given to the domain defined by the axioms of mathematical number theory. Similarly, mathematical type "string of integer" might be the name given to the domain defined by mathematical string theory, which is a generic theory in the sense that it is parameterized by another mathematical type. Note that a type is not a set of values, but rather the name of a set of values. The distinction is subtle, yet is important for future discussions.

A mathematical variable is a symbol that stands for some value from the domain of the variable's type. For example, if "x" is a mathematical variable of mathematical type integer, then x stands for some value from the domain of integer.

Mathematical types are defined and discussed completely in terms of mathematics with no mention of programs. The values which form their domains are entirely abstract. We say things about such values by writing assertions in some formal language, such as predicate calculus. For instance:

```
∀ x : integer, x + 1 > x
```

says that the integer obtained by adding one to any integer is larger than the original value.

---

17 Some might consider the operations of the type definition as the "axioms" defining the type, and argue that this is therefore an "abstract" definition. This might be true, provided the operations were formally specified, which is not done in the examples and discussion presented in the literature. Also, formally defining a type this way suffers from the same problems inherent to algebraic specifications, discussed in Section 2.4.2.2.

A program type is a name given to a program domain, which is also a set of values[18]. Each element in a program domain has a concrete representation (which may be as low-level as a configuration of bits in memory) and is modeled by an element from a corresponding mathematical domain. In other words, there is a total (mathematical) function from a program domain to a corresponding mathematical domain. The values in a program domain are defined by an associated set of operations, which is discussed in Section 3.4.

As with mathematical variables, a program variable is a symbol that stands for some value from the domain of the variable's type. A program variable is declared with the notation "x : T" to say x is a variable of type T, and consequently x stands for some value from the domain of T. However, because every value in a program type's domain has a mathematical model, it is possible to *reason* about the value of a program variable as a mathematical value, rather than as a concrete representation. In other words, the value of a program variable in a client program is considered to be the abstract value of the mathematical model of that variable's concrete representation.

For example, program type "int" might be modeled as a mathematical integer, and a client of int reasons about ints as if they were mathematical integers. Similarly, program type "stack" might be modeled as a mathematical string, and a client of stack reasons about stacks as if they were mathematical strings[19]. If a client declares "i" as a variable of type int, i should be treated as a symbol that stands for a mathematical integer. For reasoning about the program, the value of i must not be thought of as a sequence of 32 bits (or whatever the concrete representation of an int), and it certainly must not be thought to be the address that contains the concrete representation. Defining program types in terms of mathematical models allows RESOLVE's types to be truly "abstract data types."

---

18 Actually, a RESOLVE program type is the name of a *marker*, which in turn maps to a program domain. The reasons for introducing markers is introduced in Section 3.4. However, for the current discussion, it is sufficient to consider a program type to be the name of a program domain.

19 Strings are used only to reason about stacks. Program stacks probably would not be implemented as strings.

2.5.3.2 Role of Types

Now that we know what types are, let's see how they are used in a program. Types play two potential roles in a programming language — syntactic type checking, and overloaded operator resolution. Syntactic type checking validates the legality of an operation invocation. In modern programming languages all parameters to operations have associated types. When an operation is invoked, the types of actual parameters are checked against the types of formal parameters. If they are not equivalent (discussed in Section 2.5.3.3), an error is generated (or in some languages the type is automatically "coerced" to another type, as discussed in Section 2.5.3.4). It is not necessary for the language to define the semantics of an operation invocation where the actual parameters are not of the correct types — this is simply illegal, and therefore has no semantic meaning. This greatly simplifies the semantic definition of a language.

Note that nothing is mentioned about when the checking occurs. If it is done by the compiler, the language is said to be statically-typed. If it is not done until execution the language is said to be dynamically-typed. This is discussed in more detail in Section 2.5.3.5.

The second role of types, overloaded operation resolution, is used to select the appropriate operation from a set of possible ones. Each operation defined in modern programming languages has an associated name and type signature, which consists of the number and types of all parameters. In many languages it is possible to declare several operations with the same name, as long as they all have unique type signatures. This is called overloading. When one of these operations is invoked, the compiler (in the case of static typing) or run-time system (in the case of dynamic typing) compares the signature of the invocation with the signatures of all definitions. If one matches, that operation is invoked, otherwise an error is flagged. For example, "+" is overloaded in many languages, standing for both integer addition and floating point addition. Overloading is a convenient way of naming related groups of operations, yet it can easily lead to complexity both for humans trying to understand a program and compilers parsing a program [MacLennan 83, Aho 86].

In RESOLVE, the only use of types (both mathematical and program) is so the structure of program statements can be checked against expected usage at compile time. Types

are not used to resolve overloaded operations for the sole reason that operations cannot be overloaded!

2.5.3.3 Type Equivalence

The previous discussion indicated that a compiler or run-time system must be able to determine whether two types are equivalent, for example when checking the type of an actual parameter against the type of the formal parameter. There are two common definitions of type equivalence, usually called name equivalence and structural equivalence. With name equivalence, two types are equivalent if and only if their names are the same. With structural equivalence, two types are equivalent if their structures are the same.

For example, consider the following type declarations in a Pascal-like language:

```
type T1 = array [1..10] of integer;
type T2 = array [1..10] of integer;
type year = integer;
type temperature = integer;
```

With name equivalence, all four of these types are inequivalent, whereas with structural equivalence T1 is equivalent to T2, and year is equivalent to temperature[20].

Both approaches have advantages and disadvantages. Name equivalence is simple to understand and implement, and provides extra protection against type errors. However, it places reliance on globally-declared type identifiers, and can be a nuisance to programmers since they often have to make up extra type names [Welsh 77]. Structural equivalence is more flexible and at first glance more intuitive. However, unambiguous definitions of it are complex and difficult to understand and implement [Welsh 77]. Also, with structural equivalence two types that are not logically equivalent (e.g., year and temperature in the above example) but happen to be declared identically are considered equivalent, and consequently some programming errors (e.g., copying a variable of type year into a temperature variable) are not detected.

Most languages use name equivalence (or some variation of it) for reasons of safety, understandability, and ease of implementation [MacLennan 83].

---

20 Good discussions of type equivalence are presented in [Welsh 77], [Pratt 84], and [Horowitz 84].

As discussed in Section 3.4, types in RESOLVE are formally defined in terms of sets and mathematical functions on these sets, and type equivalence is defined in terms of mathematical functions mapping to the same element. Essentially, though, program type equivalence is name equivalence, and mathematical type equivalence is structural equivalence.

2.5.3.4 Type Coercion

In some languages it is possible to violate the normal type equivalence rules and consider a value of one type to be a value of an inequivalent type. This ability is called type coercion. There are two common ways for types to be coerced — implicit conversion, and representation interpretation. With implicit conversion, an operation is implicitly invoked that converts the value from one type to another. A common example of this is the implicit conversion of numeric data from one representation to another (e.g., "integer" to "real"), permitting a programmer to write "mixed expressions." Although this might reduce the number of keystrokes required to construct a program, there are several inherent dangers. First, mistakes made by the programmer may go unreported, or may be reported at some other location. For example, consider a hypothetical language where booleans are implicitly coerced to integers if necessary. Suppose a programmer needs to place the sum of integers j and k into integer i, but instead of entering the statement "`i := j + k`" he or she accidentally enters the statement "`i := j = k`" (an easy mistake on keyboards where "+" is a shifted "="). In the hypothetical language, this error will not be detected, because the language will implicitly coerce the boolean expression "`j = k`" to an integer.

A second problem with implicit coercion is that the programmer may not be aware of costly conversions that are taking place as a result of coercion, making programs very inefficient. COBOL [COBOL 74] is a language which supports many forms of implicit coercion and encourages their use. For instance, consider the statement "`ADD A TO B`" where A is an integer represented as a BCD character string and B is an integer represented as a 2's-complement binary value. This statement implicitly converts A to a 2's-complement integer before performing the addition. The conversion is much more costly than the addition. There may be no way around this conversion. However, it may be possible for the programmer to reduce the number of necessary conversions, but he or she may not realize that conversions are taking place because they are implicit.

The other method of type coercion — representation interpretation — allows the representation of a value (i.e., the bit sequence) to be interpreted as a value of another type. For example, the statement "`i = (int)j + (int)k`" in C [Kernighan 78] or C++ [Stroustrup 86] causes the representation of variables j and k to be treated as if they were ints for this statement, regardless of the types of j and k. Conversion does not take place, and thus there is no inherent inefficiency. However, this relies on the programmer having complete knowledge of the representations of the types being coerced, which is a blatant violation of information hiding. If, for example, the representation of a type is changed, coercing the value may be completely bogus.

Languages that do not include coercion mechanisms are often said to be strongly-typed languages because the only values allowed in a particular context are those whose type is equivalent to what is expected.

From the standpoint of reusability, coercion is bad. The first form encourages programs that are inefficient, and the second form violates information hiding and complicates formal specification and verification. For these reasons, RESOLVE does not include any form of coercion.

2.5.3.5 Dynamic vs. Static Typing

In some languages (e.g., APL [Iverson 62], SNOBOL4 [Griswold 71], and Smalltalk-80 [Goldberg 83]) the types of variables mentioned in a statement are not known until that statement is actually executed. For example, the types of variables x and y in the expression "x + y" may change every time the expression is evaluated. Under these circumstances the responsibility for type-checking is forced on to the run-time environment because the types of variables cannot be determined by the compiler. Languages exhibiting these characteristics are said to be dynamically-typed languages, and variables are often said to be typeless since they have no fixed type.

The advantages of dynamic typing are that the programmer is freed from the nuisance of declaring variables, and there is much flexibility in that the data object associated with a variable name may change as needed during program execution. There are also several disadvantages. First, dynamic typing is expensive in terms of space and time, since it requires overhead to store type information at run-time (i.e., "tags"), and requires code to check the types of parameters prior to every operation invocation. Second, formal

specification and verification is difficult because nothing is known about the value of any variable prior to execution. Program debugging is difficult because many errors are not detectable until run-time. Also, it is quite likely that some statements were not adequately tested, and type errors associated with them are not discovered until much later when the program is in production.

In statically-typed languages (e.g., Pascal and its descendants), the programmer meticulously associates type information with every variable and operation parameter. The compiler checks each actual parameter to an operation against the expected type, and flags an error if the types are inequivalent. There are several advantages to this scheme. First, many programming errors are detectable at compile-time, which is preferable to waiting until run-time [Hoare 83]. Second, there is no need to keep type information around at run-time and no need to dynamically check the types of parameters. Thus, statically-typed language have no run-time inefficiencies associated with type-checking. Third, it is possible to formally specify and verify programs written in statically-typed languages.

Dynamically-typed languages are not conducive to designing reusable parts, because programs in them are inefficient and very difficult to formally specify and verify. RESOLVE is therefore a statically-typed language.

2.5.3.6 Type Initialization and Finalization

In many programming languages, a variable's value is unknown (or worse, possibly unreferenceable) when the block of code containing its declaration begins execution. These so-called uninitialized variables cause headaches for both programmers and verifiers. Errors resulting from referencing a variable before a known value is placed in it are often difficult to find, especially when the behavior of the program is unpredictable as a result of the initial "garbage bits" in the variable. Particularly insidious errors are likely to occur if the variable's representation involves pointers.

The possibility of uninitialized values also significantly increases the complexity of the formal semantic definition of a language, thus working against formal specification and verification. For example, one useful component of many formal systems is the specification of an invariant for a type, which is an assertion about the value of a

variable of that type that is true at all times. An initialization assertion becomes the base case for an inductive proof that the invariant holds at all times.

The problem is not with the notion of unknown values. In fact, saying that the value in a variable is unknown is a perfectly reasonable thing to say, as long as the variable is known to contain *some* legitimate value from the domain of its type. (Of course, the unknown value must satisfy the invariant, if any, and the program's behavior should not depend upon this unknown value.) Rather, the problem crops up when the bits in a variable may not represent *any* value from the domain of the its type (e.g., before memory is allocated for a representation that involves pointers).

Most languages require the programmer to explicitly initialize all variables. Other languages (e.g., C++ and Euclid) allow types to have associated initialization routines that are automatically invoked when a variable of that type is declared. Although this latter approach is a step in the right direction, it does not prevent problems resulting from uninitialized variables because initialization is "voluntary" rather than required of all types. Indeed, what is needed to solve these problems is to have initial values specified for all types, and initialization routines that are automatically invoked when variables come into existence. This approach lends itself to formal specification and verification (and hence to design and implementation of more reusable components) because every variable is guaranteed to have a value meeting the initial specification for its type before any statement is executed.

Some might argue that requiring every variable to be initialized is too costly, and thus not conducive to software reuse. There are three reasons this is not generally true. First, a type's initial value is chosen by the part's designer, and should be one that is easy to construct (e.g., an empty stack). In fact, experience has shown that it is possible to define initial values that require a constant amount of time to construct for a wide variety of reusable components [Harms 89a]. Second, it is possible to implement "lazy initialization" where a variable is not actually initialized until it is referenced for the first time, which is discussed in Section 3.8.2. Finally, of course, even if initialization is not automatic it is nonetheless mandatory for most complex types because "garbage bits" cannot always be interpreted as a legitimate value of the type in question. Manual initialization saves nothing in these cases.

Variables whose representations involve dynamically-allocated memory should have that memory released when it is no longer needed. One way to reclaim memory is to rely on automatic garbage collection by the run-time system. Another way is to explicitly finalize every variable at the end of the block in which it is declared. Garbage collection has the advantage that it is transparent to the programmer. However, it has the disadvantage that the user may notice significant performance degradation at seemingly random (and possibly inopportune) times while the garbage collector examines and/or compacts and/or defragments memory. Using garbage collection, it is not at all easy to characterize the performance of operations because the garbage collector may execute almost any time. The problem is that the garbage collector is part of the system, and has no knowledge about the specific application or memory utilization patterns for specific reusable parts. Writing a general garbage collector that does not occasionally cause the computer to "go out to lunch" is not possible.

Another approach is to define a finalization routine for each type where dynamic memory must be reclaimed, and which is automatically invoked. It would seem that the performance of finalization routines would not be very good, since they must be able to finalize whatever structure is left in a variable (e.g., empty stacks as well as stacks containing one million items). Surprisingly, our research has shown that it is possible to design representations such that initialization and finalization require a constant amount of time, even for complex types such as arrays and lists [Harms 89a]. This is due in part to the fact that the finalization routine has knowledge about how the various parts of the representation interact, something a garbage collector would have no way of knowing.

So, a language supporting the design of reusable software should require every type to have an initial value and a corresponding initialization routine. In addition, it should allow an implementer to write a finalization routine if dynamically allocated memory is to be released. Both of these routines should be automatically invoked. Initialization is necessary largely to support formal specification and verification, and finalization is required for efficiency and performance characterization. As explained in Chapter 3, this is the approach taken in RESOLVE.

2.5.3.7 Built-In Types

Almost every programming language has a number of types and type constructors defined in the language (e.g., integer, character, boolean, array, and record). There are several reasons why these types are built-in. First, simple types and constructors have been defined in languages since the early days of FORTRAN, lending historical credibility to the notion of built-in types. It is interesting to note that in languages such as FORTRAN, these simple types were the *only* types available to the programmer, and these languages were consistent in the sense that *all* types were built-in. This of course is not true of modern languages where programmers can define their own types.

A second rationale for defining built-in types is that it makes the task of constructing a program somewhat easier. For instance, many programs need these types, so why force the programmer to explicitly declare them? Also, accessing built-in values is often more convenient than accessing user-defined structures because languages provide special syntax for built-in types, such as infix notation (e.g., x + y) and array subscripting (e.g., a[i]).

Third, there is the perception that it is more efficient to implement simple types within the language, as opposed to letting the programmer implement them. The confusion here is between specifying a type and implementing it. Obviously some types (e.g., integer in the standard representations) have very efficient implementations that cannot be expressed in the high-level language. However, this does not justify building the specification of the type into the language.

Finally, most languages have control structures, such as if and while statements, whose execution depends on some value of a particular type (e.g., integer or boolean), which therefore must be defined in the language. It is possible, however, to define all control structures in such as way that they do not rely directly on data values. This is the approach taken in RESOLVE, discussed in Section 3.3.2.

The conclusion from this discussion is that there are no overwhelming technical reasons for including built-in types in a language. But what harm can they do — specifically, do they discourage reusable software? The answer is that built-in types do have several disadvantages, and they do actually discourage reusable software design. For instance, built-in types are usually treated differently than other types, as discussed previously

(i.e., the programmer doesn't declare them, and special syntax is used to access their values). In fact, built-in types in some languages are conceptually very different from other types. For example, all types in Eiffel are objects in the typical objected-oriented sense *except* the built-in types which are considered as types in traditional nonobject-oriented languages. The inconsistency between built-in types and user-defined types increases the complexity for a programmer learning the language and for its formal semantics. A better alternative is to have all types uniformly declared and referenced via identical mechanisms.

A second disadvantage of built-in types is that multiple implementations are not available, and the programmer is required to use whatever representation is defined in the language implementation. Specifically, the programmer cannot choose the most appropriate implementation from an implementation library[21], which violates one of the characteristics of reusable parts — multiple implementations — discussed in Section 2.3.4.

A third disadvantage is that built-in types must be explicitly incorporated into the proof rules and formal semantic description of the language, thus complicating the formal language description. An approach that simplifies the language description uses the same language mechanism to define *all* types.

RESOLVE has no built-in types, and is similar to Alphard [Shaw 81] in this respect. The specific implications of this are discussed in Chapter 3.

2.6 Programming Language Survey

In Section 2.3 well-defined reusable parts were characterized, along with a discussion of how these characteristics affect programming language design, and brief evaluation of features of existing languages with respect to these characteristics. The conclusion was that no modern language has all of the necessary constructs that encourage and facilitate the design and implementation of reusable software components.

---

21 Some languages (e.g., C, C++, and Ada) define several flavors of integers (e.g., short, int, and long). The differences between these are not defined or characterized in the language, but are instead defined in each language implementation. Also, their distinctions are usually described in implementation-level terms such as "bits" and "2's-complement," which violates the principle of information hiding. Thus, this approach does not adequately address the concerns raised in the current discussion.

Despite this conclusion, it is worthwhile examining several important programming languages to see exactly why they are not as appropriate as they could be for development of reusable software. Sections 2.6.1 through 2.6.4 examine several Pascal-like languages, namely Ada and Anna, Modula-2, Euclid and Gypsy, and Alphard. In the following three sections several "object-oriented" languages are examined — C++, Eiffel, and Larch/CLU. The last section examines Z, which is an implementation language-independent specification language.

2.6.1 Ada and Anna

Ada [DoD 83] is the quintessential product of "design by committee." The U.S. Department of Defense developed the specification for the language in several stages (called Strawman, Woodenman, Tinman, Ironman, and finally Steelman) over several years. The actual language was selected from a group of sixteen submitted in an international competition. Reviews were held at each stage, allowing many people the opportunity to influence the design and selection of the language.

As a result, Ada is a "jack of all trades." It is a general-purpose language based on Pascal, it is a real-time language for such applications as guided missiles, and it is a systems programming language. It has facilities to model parallel tasks and handle exceptions, and allows access to system dependent parameters and precise control over the representation of data. In addition it encourages programs to be designed using several software engineering principles such as modularization and information hiding.

Unfortunately, Ada does not include the constructs to formally specify the intended behavior of a program. Anna [Luckham 87] is a language extension to Ada that partially addresses this void. It is designed to "meet a perceived need to augment Ada with precise machine-processable annotation so that well established formal methods of specification and documentation can be applied to Ada programs" [Luckham 87]. Annotations can supposedly be used by a verifier to formally verify a program, and many of them can also be used to generate run-time consistency checks. The extensions to Ada are in the form of formal comments (sometimes called virtual Ada text), so Ada is a proper subset of Anna in the sense that any valid Anna program is also a syntactically-correct Ada program[22].

---

22 Because of this relationship, any reference to Ada in this discussion applies to Anna as well.

The basic encapsulation mechanism in Ada (Anna) is the *package* in which types, constants, and operations are defined for access by a client. Packages may be generic in the sense that they are parameterized (see Section 2.3.3). There is no notion of an inheritance hierarchy, so package design generally follows the abstract data type approach rather than the abstract data object approach (see Sections 2.4.1.1 and 2.4.1.2).

A package "specification" in Ada is no more than operation signatures. However, Anna packages can be formally specified by defining axioms and other assertions, thus encouraging formal specification using the algebraic approach (see Section 2.4.2.2). The specification of a package can be separated from its implementation (called the *package body*), allowing the implementation of routines and the representation of types (called *private* types) to be hidden from clients. However, to simplify compilation of clients, the representation of a private type must be defined within the package specification, even though a client cannot reference this information. Thus, all clients must be recompiled every time the representation of a private type changes. Also, Ada is designed to have one package body per package, so multiple implementations of a package are not encouraged, though it is possible through some sequence of awkward contortion in an Ada programming environment. Even in this case, though, a client program must have one package body for every instance of that package, so only part of the flexibility of multiple implementations can be achieved.

As discussed in Section 2.5.2, the meaning of assignment and equality operations on private types depend upon whether a type uses pointers in its representation. Ada's solution to this ambiguity involves the notion of a *limited private* type on which the use of the predefined operators for assignment and equality testing is not permitted. In other words, a client cannot assign a value to a variable whose type is limited private.

The built-in operators available on private types are assignment and equality check. These operators are not available on limited private types, providing a solution to the problem of interference between hidden types and pointers discussed in Section 2.5.2.

With these features, Anna appears to be a language encouraging the design and implementation of reusable parts — a part can be formally specified, the specification can be separated from its implementation, and generic parts can be defined. Unfortunately, the keyword here is "appears." In fact, Anna has some serious flaws

making it unsuitable for designing reusable parts. For example, it is not at all easy to define multiple implementations of a specification, and many problems of efficiency discussed in Sections 2.3.5 and 2.5.2 are not adequately addressed.

However, the biggest problems inherent to Anna are correctness and specification problems. For example, because formal assertions can be incorporated into an Anna program, they appear to be formally specified and verifiable. However, the definition of Anna does not include the proof rules, axioms, and semantic definitions that would be necessary to formally verify Anna programs. The authors allude to this omission, claiming that these would be similar to those developed for Pascal (i.e., [Hoare 73] and [Luckham 79]), and therefore almost trivial to develop. However, Ada is significantly more complex than Pascal, with many of the same features that complicate Pascal's proof system (e.g., implicit aliasing). Its proof system would not be a trivial extension to Pascal's, and from all indications has not been developed. It appears that Anna is actually oriented toward run-time checking rather than formal verification, despite early claims to the contrary.

Another problem with Anna is that many potential errors are not easily checkable. For example, the order in which actual parameters are evaluated is not defined, and it is an error for a program to take advantage of any particular evaluation order. However, this error is not detectable by the compiler, nor is it possible to write Anna assertions that detect it.

Finally, there are no restrictions placed on actual parameters, causing a problem when the same identifier appears more than once in an argument list. For example, consider the following Ada procedure:

```
procedure pathology (x,y : in out INTEGER) is
begin
  x := x + 1;
  y := y + 2;
end pathology;
```

If this procedure is invoked with "`pathology(b,b)`" the value in b upon return has either been incremented by 1, 2, or 3, depending upon the order of parameter evaluation

and whether **in out** is implemented as value/result or reference. This is clearly not a good idea[23].

To summarize, Anna supports and encourages the development of formally specified generic packages. Facilities are provided to separate the implementation of a part from its specification. However, Anna does not easily support multiple implementations for a specification. Also, Anna includes features such as implicit aliasing which make it extremely difficult to develop a proof system. In fact, there are no proof systems for Anna, and until one is developed it is not possible to formally verify Anna programs. Also, Anna does not provide alternatives to the problems inherent to copying. The conclusion is that Anna does not provide all of the features necessary to support and encourage the design and implementation of reusable software components.

2.6.2 Modula-2

Modula-2 [Wirth 82] is a programming language that extends Pascal with concepts for modular program organization, multiprogramming, and access to low-level machine facilities. It is intended to be a general-purpose language encouraging structured programming and permitting the development of "system" programs (e.g., to control hardware devices such as printers).

Modula-2 programs are organized into units called *modules* that encapsulate data type definitions and operations with parameters of those types. This suggests the development of modules designed around the abstract data type approach (see Section 2.4.1). Modules control which identifiers are available to clients using explicit export lists, and clients indicate which identifiers they need using explicit import lists. Modules cannot be parameterized, so it is not possible to define generic modules.

It is possible to divide a module declaration into two parts — a *definition module* containing the signatures of exported types and operations, and an *implementation module* containing data structures and code implementing those types and operations. Each definition module has exactly one implementation module, so it is not possible to have multiple implementations of a definition. A client program can access only those

---

23 It is interesting to note that using the same variable more than once in an actual argument list was prohibited in the preliminary definition of Ada ([Ada 79] and [Ichbiah 79]), yet is not prohibited in the final definition.

items defined in the definition module. Similarly, a client programmer should not need access to anything in the implementation module. Thus information hiding is enforced for both client programs and client programmers.

A type defined in a definition module (i.e., exported) may be either transparent or opaque. A transparent type is completely defined in the definition module and clients can directly access its representation. On the other hand, the representation of an opaque type is defined only in the implementation module, and clients cannot access its representation. To simplify compilation of clients, the representations of all opaque types must occupy a fixed amount of storage — specifically, the amount of memory necessary to store a machine address (i.e., a pointer). This implies that most opaque types are represented as pointers to representation structures. Unfortunately, a client has no way of knowing this for sure, leading to the ambiguities discussed in Section 2.5.2.

It is important to note that a definition module does *not* contain a formal specification of the module, but merely signatures of operations and (hopefully) informal descriptions within comments. In fact, formal specification was not a design goal of Modula-2, and no facilities are provided for writing specifications.

In summary, Modula-2 provides the mechanisms to separate (informal) specification from implementation, which is a significant improvement over Pascal. However, it does not address the issues of formal specifications and formal verification, it does not permit the declaration of generic modules, it does not permit multiple implementations of a specification, and it does not provide any alternatives to the problems inherent to copying (discussed in Sections 2.3.5 and 2.5.2). Under the criteria for reusable parts developed in Section 2.3, the conclusion is that Modula-2 does not support the design and implementation of reusable software parts.

2.6.3 Euclid

Euclid [Lampson 77, Lampson 81] is a programming language “evolved from Pascal by a series of changes intended to make it more suitable for verification and for systems programming” [Popek 77]. These goals unfortunately conflict with each other, and the language is therefore a compromise between them.

The encapsulation mechanism is the *module*, which is similar to a record (i.e., it is defined as a "type"), with the possibility of type and operation fields. Modules (actually, any type declaration) may have parameters, thus allowing the definition of generic parts. The identifiers (i.e., fields) that are available to a client are explicitly listed in an export list, allowing the compiler to enforce information hiding from the client program. If language-defined assignment and/or equality checking are to be allowed on variables of an exported type, these operators must be explicitly exported with the type.

Initialization and finalization routines can be defined for a module, which are automatically invoked for every variable of the *module* type. Unfortunately, it is not possible to define initialization or finalization routines for types provided by the module. Thus, module initialization/finalization is supported, but not data type initialization/ finalization.

A module is formally specified using assertions that specify invariants and pre- and post-conditions[24]. There are neither mechanisms for defining axioms for a type, nor for modeling a type as a complex mathematical type (e.g., string). Rather, types are defined in terms of mathematical models of the built-in types (i.e., Integer[25], Boolean, and Char) and type constructors (e.g., record, array, and pointer). In other words, types are defined in terms of implementation structures, and therefore should not be considered "abstract" data types.

Formal specification of the items defined in a module are placed with the definition of each item, so there is no attempt to separate specification from implementation (i.e., the module is not divided into separate "specification" and "implementation" parts). The implications of this are that 1) it is not possible to have multiple implementations of a specification, and 2) it is not possible to enforce information hiding from a client programmer.

Both explicit and implicit pointers are defined in Euclid. The formal specification of explicit pointers (and explicit aliasing) is similar to the proof system presented in [Hoare 73] and [Luckham 79]. To make verification somewhat easier, Euclid

---

24 Although formal specification and verification are primary goals of the language, none of the sample programs in [Lampson 77] or [Lampson 81] are formally specified!

25 Type Integer is strictly a mathematical type, and it is not possible to declare variables of type Integer. All "integer" variables must be declared as a subrange of Integer. Two Integer subranges — SignedInt and UnsignedInt — are predefined for convenience.

introduces the notion of a collection. Each collection has an associated type, and pointers are declared to be elements of a collection rather than pointers to a type. A pointer from one collection cannot be assigned to a pointer from another one, even if the collection types are the same. Thus, pointers from different collections cannot alias the same location, and this knowledge can simplify verification. For example, consider the following code segment:

```
...
{assert ...}
x^ := 3;
y^ := 5;
{assert x^=3 and y^=5}
```

If x and y are pointers from the same collection, they may alias the same location. Therefore the condition "x ≠ y" must be proved for the code to be valid. On the other hand, if x and y are from different collections, they cannot alias each other, so nothing additional has to be proved. Thus, collections allow a programmer to partition pointer variables to indicate some of the knowledge about how they will be used.

A more significant simplification for the verifier is that implicit aliasing cannot occur. This is enforced by not allowing a location to be passed as an actual call-by-reference parameter if it could be accessed by another name within the routine. For example, the invocations "`proc1(a,a)`" and "`proc2(b,b[2])`" are illegal if both parameters are call-by-reference. The invocation "`proc1(b[i],b[j])`" requires the assertion "i ≠ j" to be proved by the verifier. Also, the invocation "`proc1(a,c)`" is not allowed if either a or c is directly accessed within the routine. This last example demonstrates that the compiler must have access to the implementation of all routines, because it must determine which non-local variables each routine references.

In spite of this, Euclid includes some strange constructs that come tantalizing close to aliasing. For example, it is possible to declare several variables as explicit aliases for one location, similar to FORTRAN's EQUIVALENCE statement. Also, it is possible to explicitly indicate the fixed memory address for a variable, which could conceivably be anywhere in memory. This latter construct makes it impossible to guarantee that implicit aliasing cannot occur, and in fact makes it impossible to develop a verification system that is independent of a language implementation.

Also, Euclid defines type equivalence to be essentially "structural" equivalence, which significantly complicates the typing system.

In summary, Euclid is an extension to Pascal that has formal specification and verification as primary goals. Formally specified types and operations can be encapsulated into modules, which may be generic. However, specification is not separated from implementation, and multiple implementations for a specification are not supported. Also, there is no attempt to provide alternatives to the problems inherent to copying. Therefore Euclid does not provide the features necessary to encourage the design and implementation of reusable software parts.

2.6.4 Gypsy

Gypsy [Ambler 77] is a Pascal derivative whose design was driven by the development of a comprehensive methodology for constructing verified programs oriented toward communications processing. The goals set by the language designers were ambitious — formal specification and verification of programs, constructs to enable development of systems programs (e.g., concurrency and synchronization structures), and the ability to support execution in imperfect execution environments (e.g., exception handlers).

Gypsy does not provide any mechanism for defining generic types nor for encapsulating types and operations into a syntactic unit. Instead of encapsulation, an access list can be associated with a type, indicating the set of operations that have access to its internal representation. For example, one could define a type stack which permits only operations Push, Pop, and IsEmpty access to its internal representation. This permits information hiding from client code to be enforced, but does not permit information hiding from a client programmer. The claim is that this scheme permits the development of abstract data types. However, with all programs coded as monolithic compilation units, this approach surely does not *encourage* the development of abstract data types.

Types and operations are formally specified by assertions such as pre- and post-conditions and invariants. These assertions are written in terms of program structures (e.g., integers, characters, arrays, and records) as well as a handful of mathematical types (e.g., sequence). The specification of an item is placed with its implementation, so a specification is not separated from its implementation. Another implication of this is that it is not possible to define multiple implementations of a specification.

Gypsy has several features that make verification easier. For instance, explicit pointers are not defined in the language, procedures cannot access non-local data, and functions cannot produce side-effects. However, it is not clear whether implicit aliasing (resulting from call-by-reference parameter passing) is possible.

In summary, Gypsy includes constructs for formal specification and verification. However, it does not have mechanisms for declaring generic components, for separating specification from implementation, for declaring multiple implementations for a specification, nor does it address the problems inherent to copying. Also, there is no encapsulation construct, so parts as separately-compilable units cannot be developed. The conclusion is that Gypsy cannot be used to design and implement reusable software parts.

2.6.5 Alphard

Alphard [Shaw 81] is a programming language designed to support data abstraction and program verification, and evolved between 1974 and 1980 by a research group at Carnegie Mellon University. Development of Alphard, at least as a focused research project, stopped around 1980. However, many ideas developed in Alphard are germane to the current discussion of reusable parts, and worth examining in some detail.

The encapsulation mechanism in Alphard is the *form*, which defines a data type along with operations having parameters of that type. A form may be generic in the sense that it may have other forms (i.e., types) as parameters. A data type characterizes the possible values a variable may have, and determines the contexts where variables can be used (i.e., type checking of actual against formal parameters). Because types are not considered to be values along with operations, module designs following the abstract data type approach are suggested (see Section 2.4.1).

The meaning of a form identifier is context-sensitive. When used in a variable declaration it stands for only the data type defined in the form. When used as an actual parameter to an instantiation of a generic form it stands for the encapsulation of its type and operations. For example, "**var** `x:integer`" declares x to be a variable of the type defined by form integer. (Note that variables are considered to be values, *not* encapsulations of values with operations.) On the other hand, "**var** `s:stack(integer)`" instantiates generic form stack, passing it form integer as a

parameter. In this example, all operations defined by form integer are passed to the instantiation of form stack. Overloading the meaning of a form identifier is unfortunate because it confuses the distinction between an encapsulation and a data type.

Language mechanisms are provided to formally specify types and operations. Types are specified using mathematical models, and operations are specified with pre- and post-conditions written in terms of functions and relations defined in mathematical theories. For example, type stack could be modeled as a mathematical string, and the effect of operation push would be specified as assertions using functions defined in string theory. This is the model-based approach to formal specification, discussed in Section 2.4.2.3. Alphard does not, however, provide mechanisms to formally define mathematical theories, and a verifier would have to obtain these definitions from someplace other than an Alphard program.

Every type has an initially assertion associated with it that is guaranteed to be met before a variable of that type is accessed for the first time — in other words, every variable has an initial value. There is an initialization routine for each type that is invoked for every variable declared of that type, and a corresponding finalization routine that is invoked when variables are destroyed. The advantages of this with respect to reusability are discussed in Section 2.5.3.6.

In addition to formal specification of the type and operations, a form contains declarations of the representation of the data type as well as implementations of all operations (including initialization and finalization). A client program can only access the identifiers defined in the specification portion of the form, and therefore does not have access to the representation of the data. However, a client programmer has access to the entire form. Thus, Alphard supports information hiding from the client program but not the client programmer.

This organization has other consequences. For example, it is not possible to define multiple implementations of a specification because specification and implementation are coupled into one compilation unit. Also, it is not possible to parameterize specification separately from implementation. Thus, for example, the parameter list of a form defining an associative memory structure may mention things like “hash function” which have nothing to do with understanding the specification. Thus, placing specification and

implementation into the same module can lead to confusion, and parts developed would not be as reusable as they might be if multiple implementations were permitted.

Another interesting characteristic of Alphard is that only two forms are built-in to the language — boolean and rawstorage. Boolean is built-in because of its relationship with some control structures (e.g., the conditional statement), and rawstorage because it is needed to bootstrap all other forms. There is, however, a "standard prelude" of forms that is automatically appended to the beginning of every compilation, which includes forms such as integer and vector. Some advantages to this approach are discussed in Section 2.5.3.7.

Despite the emphasis on formal specification, there are a number of features of Alphard that work against verification. For instance, restricted use of aliasing is part of the language definition. An example of restricted aliasing is that actual parameters cannot alias a location (see the example in Section 2.6.1), except if the formal parameter is specified with the qualifier 'alias.' It was felt that placing appropriate restrictions on aliasing was safe enough to warrant its inclusion, and that the impact on the proof system would be minimal. Unfortunately, this was not the case, and the inclusion of aliasing contributed to the inability to develop a formal proof system for Alphard [Shaw 81].[26]

Another feature that complicates the language (and development of a proof system) is the horrendously complex type matching scheme. This complexity is partly the result of the capability to place restrictions on forms that are passed as actual parameters to instantiations of generic forms. For example, it is possible to require the actual form to have an assignment operation defined, or for that matter *any* set of operations meeting arbitrary specifications.

Finally, note that Alphard makes no attempt to provide alternatives to copying large structures, except the usual reliance on pointers.

In conclusion, Alphard was designed to encourage and promote the development of formally specified and verifiable modules defining abstract data types. Unfortunately,

---

[26] It is interesting that no one realized that explicitly allowing aliasing of actual parameters wouldn't complicate the proof system, and equally surprising that after the Alphard experience was reported, later specification/verification systems persisted in embracing aliasing.

the language became so cumbersome and complex that this goal was not achieved, although significant progress was made toward it.

2.6.6 C++

C++ [Stroustrup 86] is an extension to C [Kernighan 78] that incorporates encapsulation of data with operations, and other constructs often associated with "object-oriented" programming languages. C++ is a superset of C in the sense that most C programs are legal C++ programs, and consequently C++ inherits (pun intended!) many of C's problems.

The encapsulation mechanism provided by C++ is the *class*, which is essentially a C `struct` extended in two ways — 1) fields (called *members*) are categorized as either *public* or *private*, and 2) members may be data or functions. A private member is accessible only within member functions, whereas a public member is accessible in both member and non-member (i.e., client) functions. In typical classes, data members are private and function members are public. This organization prevents a client program from directly accessing the representation of a class, thereby enforcing information hiding from the client program. However, the client programmer has access to the private members of the class, so information hiding is not enforced from the client programmer.

Note that the members of a class are data and functions. Specifically, a class cannot have a type member. The encapsulation approach suggested is therefore the abstract data object approach, discussed in Section 2.4.1.1.

The definition of a member function in a class definition includes the function's signature and optionally the function's code. Notably lacking from the definition is any kind of formal specification. Indeed, neither formal specification nor formal verification are mentioned in [Stroustrup 86], and it is safe to conclude that formal methods played no role in the design of C++.

It is possible to separate "specification" (i.e., function signatures) from implementation using the header file mechanism of C++. In this approach, the definition of a class is contained in a header file, and does not include the code for member functions. A client of the class only needs access to this header file. The code for the member functions is

contained in a separately compiled file. For example, a header file (e.g., IntStk.h) for class IntStk might contain:

```
class IntStk {
  int arr[100];         // representation of integer stacks
  int top;
public:
  IntStk ();            // constructor
  void Push(int x);     // add integer onto the stack
  int Pop ();           // remove the top integer
  int IsEmpty ();       // return true iff stack is empty
};
```

A separately compiled file would contain the implementation of the member functions:

```
#include "IntStk.h"

IntStk::IntStk ()
{ top = 0; }

void IntStk::Push (int x)
{ arr[++top] = x; }

int IntStk::Pop ()
{ return arr[top--]; }

int IntStk::IsEmpty()
{ return top==0; }
```

A sample client of IntStk is:

```
#include "IntStk.h"
...
IntStk s1,s2;
int x,y;
...
s1.Push (y);      // push y onto stack s1
x = s2.Pop ();    // pop the top item from s2 into x
```

This example suggests and demonstrates two important characteristics of C++ classes. First, each class has exactly one representation (declared as private members of the class), and each member function, like every C++ function, has exactly one implementation. In other words, C++ does not directly support multiple implementations for a specification, although derived classes can be used to effect some aspects of this. Another implication is that if the representation (i.e., a data member) of a class changes, *all* clients of that class must be recompiled, even though this information is contained in the private portion of the class and is not accessible by the clients.

Second, it is possible to define constructor functions (and a destructor function) for a class. The constructor function has the same name as the class (e.g., IntStk) and is automatically invoked on each variable of the class at the beginning of a block containing the variable's declaration. A constructor may have parameters, in which case actual parameters are placed in parentheses following the variable's name. A destructor function (not demonstrated in the example) has the name of the class preceded by a tilde (e.g. ~IntStk) and is automatically invoked on each variable of the class at the end of a block where it is declared.

C++ allows a class to be *derived* from another class (called the *base class*) — a mechanism called *inheritance* in most object-oriented languages. The derived class inherits all members of the base class (though it only has access to the public members), and it optionally adds members of its own. Because of this relationship, every variable of the derived class is also considered a variable of the base class. The converse is not true, however (i.e., a variable of the base class is not considered to be a variable of its derived classes). A derived class extends only one base class, so C++ supports "single inheritance."

A derived class can redefine (i.e., overload) functions defined in the base class. Redefined functions are not required to have the same signatures as the functions in the base class. If a derived class redefines all functions from its base class, it has essentially reimplemented the base class. This is a very restricted form of multiple implementation of a specification, but is the only one available to C++ programmers.

A base class may define some of its member functions as *virtual* functions, which derived classes may redefine. A virtual function redefined in a derived class must have the same signature as the virtual function in the base class. There is a difference between overloading a function defined in the base class and redefining a virtual function — binding of overloaded functions is done statically at compile-time, whereas binding of virtual functions is done dynamically at run-time.

C++ does not directly support the specification of generic classes. However, some characteristics of generic classes can be effected by defining a macro (i.e, `#define`) that, when invoked with actual type(s), defines a new "non-generic" class. This is only practical if the implementation is not separate from the specification, and the allowable

actual parameters to the macro are restricted to all be the same size (e.g., pointers). This may appear to be somewhat convoluted, which it is.

A feature of C++ inherited from C is its extreme reliance on explicit pointers. Not only can a pointer variable contain the address of dynamically-allocated memory, it can also contain the address of a statically allocated variable. In addition, "pointer arithmetic" can be performed on pointer variables. These features (along with many others) make a modular proof system for C++ virtually impossible to define. Consequently there is little hope that large C++ programs could ever be formally specified or verified.

In summary, C++ is a language that does not adequately meet any of the criteria for reusable software development. It is not possible to formally specify or verify C++ programs. The "specification" of a part can be separated from its implementation, but this separation is not encouraged. Through a complex system of macro definitions it is possible to define a very restricted generic class, which is cumbersome and not the suggested approach. Likewise, it is possible to use inheritance to permit multiple implementations of a specification, but this is not encouraged. Finally, C++ offers no alternative to the problems inherent to copying, except reliance on explicit pointers. In conclusion, C++ cannot be used to design and implement reusable software components, as defined in Section 2.3.

2.6.7 Eiffel

Eiffel [Meyer 88] is an "object-oriented" programming language designed to encourage the development of reusable software components that are both correct and efficient. With these goals, it is not surprising that many of the issues raised in this dissertation are addressed by Eiffel. What is interesting, though, are the significant differences between the approach taken by Meyer in designing Eiffel, and the approach taken in RESOLVE (discussed in Chapter 3).

In Eiffel, items (called *features*) are encapsulated in a structure called a *class*, which may be generic. There are two kinds of features — data (called attributes) and operations. Features (both attributes and operations) that can be directly accessed by a client are explicitly exported from the class. All non-exported features are inaccessible by the client, enforcing information hiding from client programs. Because data (rather than

types) are encapsulated in classes, Eiffel modules are designed using the abstract data object approach, discussed in Section 2.4.1.1.

A type in Eiffel is either simple (i.e., integer, real, character, or boolean) or a class. Structured types, such as array, are predefined classes from a system library. All simple types are built-in to the language, and are defined in the abstract data type style (see Section 2.4.1.2) — the language defines a type and a set of operations with parameters of that type, and a variable of a simple type is considered to contain a value from its type's domain. Classes, on the other hand, are defined in the program (or the system library) using the abstract data object style — they encapsulate data with the operations on that data. Furthermore, a variable of a class type is considered to contain a pointer to the class' representation (i.e., data and operations). An Eiffel programmer must consequently alternate between the abstract data type and abstract data object styles of programming.

Every variable contains a known value at the beginning of the block in which it is declared. The value is determined by the variable's type — integers contain zero, booleans contain false, characters contain the null character, reals contain 0.0, and classes contain a void reference (recall that class variables are considered to be pointers). An actual object is created by executing the Create operation on the class variable, which allocates memory for the object and initializes its attributes. A class may explicitly define a Create operation to initialize its attributes. Otherwise, all attributes are initialized to the appropriate known value, as discussed above. Memory occupied by unreferenceable objects is reclaimed by automatic garbage collection. Thus, Eiffel addresses some of the problems discussed in Section 2.5.3.6 concerning uninitialized variables, but automatic user-defined finalization is not supported.

It is possible to have more than one class variable reference a given object. This potential for (explicit) aliasing complicates the formal specification and verification of a program, for reasons discussed in Section 2.5.1. Of more significance, though, is the fact that implicit aliasing can occur as a result of parameter passing, significantly increasing the complexity of the formal proof system, as discussed in Section 2.6.1.

Eiffel includes syntactic slots for placing assertions defining such things as pre- and post-conditions, module-level invariants, and loop invariants. However, formal specification was not the primary motivation in the design of Eiffel, and Meyer admits

that Eiffel is not powerful enough to formally define reusable parts. For example, assertions must be boolean expressions that can be evaluated at run-time. The assertion language does not include the means to express universal or existential quantification, making it so weak that even the simplest reusable component — the LIFO stack — cannot be formally defined [Meyer 88]. Thus, assertions are used more as a debugging aid than for formal specification.

Also, Eiffel assertions are written in terms of the component's state, which consists of the values of all attributes defined within the class. Abstraction cannot be utilized in defining a component, since the definition must be made in terms of implementation-level structures (i.e., attributes). This restriction limits the usefulness of assertions for specifying the behavior of a component.

The specification and implementation of a class are defined together, and information hiding from a client programmer is not enforced. In fact, Meyer argues that the implementation of a class should not necessarily be kept secret from a client programmer. However it is possible to have the environment produce a "short" version of a class containing only the public information.

Eiffel supports and encourages development of classes that inherit one or more other classes. A class that inherits the features of another class is called an "heir" to that class, and the class that is inherited is said to be its "ancestor." A class is compatible with all of its ancestors (e.g., a variable of class A that is an heir of class B may be assigned to a variable of class B). Eiffel supports multiple inheritance because a class can have multiple ancestors.

Eiffel uses inheritance in three ways — class extension, class modification, and multiple implementations of a class. In class extension an heir adds attributes and/or operations to those defined in the ancestor. An heir may also modify an ancestor by reimplementing some of its features. However, the modifications must meet the specifications defined in the ancestor (e.g., a redefined operation must have the same signature and meet the pre- and post-conditions specified in the ancestor).

Finally, a class can declare one or more features as *deferred*, which must be implemented by heirs of the class. The specification of the deferred feature is defined in the ancestor, and all implementations of that feature must meet this specification. A

class that has one or more deferred features is called a deferred class. Variables of a deferred class can be declared, but it is not possible to Create an object of a deferred class. Deferred classes provide a mechanism for realizing a limited form of multiple implementations for a specification — the ancestor class contains the specification, and heirs contain the implementations. The problem with this approach is that a client programmer changes implementation of an object by changing that object's class. The fact that the new class is a different implementation is only apparent from examining the class hierarchy.

Eiffel takes an interesting approach to information hiding between a class and its heirs — a class has access to *all* features of its ancestors. In other words, there is no information kept secret between a class and its heirs. The argument for this is the flexibility permitted by allowing an heir to "reuse" and extend portions of an ancestor's implementation in ways not necessarily foreseen by its original designer. Of course, an obvious implication is that when an implementation of a class changes, *all* heirs must be examined and possibly modified — a formidable task at best.

In summary, Eiffel is a language that encourages the development of software modules (called classes) using the abstract data object approach to encapsulation. Classes may be generic, and it is possible to specify a class. However, the specification language is not powerful enough to allow complete formal specification, which is rather unfortunate because many of the right ideas are addressed — for example, an operation that is redefined in an heir must still meet the specification given in the ancestor.

It is not possible to declare explicit pointer types, but class variables are implemented as pointers to an object, which is necessary to know in order to understand the language. Also, both explicit and implicit aliasing are permitted, which would complicate a proof system if Eiffel had one.

Information hiding is enforced only between a class and a client program. By design, information hiding is not enforced between a class and a client programmer, nor between a class and its heirs. The inheritance mechanism can be used to separate specification from implementation and to have multiple implementations of a specification. However, this is a somewhat cumbersome way to accomplish these objectives, and in fact, neither objective is seen as a particularly interesting or worthwhile goal in and of itself. Finally, Eiffel does not provide any alternative to the

problems inherent to copying, except for pointers. The conclusion is that Eiffel does not have the necessary mechanisms to encourage the design of reusable software components, even though this is one of its primary design goals.

2.6.8 CLU and Larch/CLU

CLU [Liskov 81] is a programming language designed to support program development by defining *clusters* that encapsulate data with operations. Though reusability was not a design goal of CLU, it provides many of the constructs necessary to design and implement reusable parts, which are discussed in this section. CLU does not provide mechanisms for formal specification and verification.

Larch [Guttag 85, Wing 87] is a family of specification languages. The Larch Shared Language is used to define mathematical theories (called *traits*) by formally defining mathematical types (called *sorts*), constants, and functions, similar to the presentation in Section 2.4.2.1. A Larch Interface Language is defined for a programming language for which programs are to be formally specified, and essentially extends the programming language with constructs necessary for formal specification. The design of each Larch Interface Language is heavily influenced by the syntax and semantics of the particular language. Specification of a program in a Larch Interface Language is written in terms of mathematical functions and relations for a trait developed in the Larch Shared Language. Larch/CLU is one such interface language, and programs written in it are formally specified CLU programs.

In order for a Larch Interface Language to be useful for formal verification, proof rules and formal semantics must be defined for it. It is not clear from the literature whether Larch Interface Languages include constructs for annotating actual code, or whether they are used solely for specifying the behavior of a program (or module). All discussion and examples presented in [Wing 87] and [Guttag 85] concentrate on specification rather than formal semantics or proof rules.

A cluster (which may be generic) encapsulates data with all operations that manipulate that data. Because data rather than types are encapsulated, CLU (and Larch/CLU) support the abstract data object approach to encapsulation, discussed in Section 2.4.1.1. However, CLU does not support cluster inheritance, and so is not considered a true “object-oriented” language.

The Larch/CLU specification of a cluster models the data as a value from a sort (i.e., mathematical type) whose trait (i.e., mathematical theory) was developed in the Larch Shared Language, and operations are specified using pre- and post-conditions that involve functions and relations from that trait. In this respect, Larch/CLU specifications follow the model-based approach discussed in Section 2.4.2.3. However, the development approach described in [Wing 87] suggests a new trait be developed for each cluster, rather than defining a cluster in terms of an existing trait. This approach has all of the problems inherent to the development of new mathematical theories, discussed in Section 2.4.2. Larch, though, does not prevent a designer from using a trait from a library (such as [Guttag 86]), and in fact this possibility is briefly mentioned in [Guttag 85].

The specification of a Larch/CLU cluster is defined independently of any implementation, and the CLU library mechanism allows several implementations to exist for a specification. Larch/CLU therefore supports both separation of specification from implementation, and multiple implementations for a specification. However, a single client cannot choose different implementations for different instances of the same specification.

CLU was not designed for formal verification, and includes some features that complicate formal specification and verification. For example, all variables are considered to be pointers to objects, and it is possible (and sometimes necessary) to alias an object by having several variables reference it. As discussed in Section 2.5.1, this aliasing complicates specification and verification.

In conclusion, Larch/CLU provides mechanisms for formally specifying a generic cluster and separating this specification from its implementation(s). A specification is developed using a two-tiered approach, with a mathematical theory specified in the Larch Shared Language, and the specification of a cluster in Larch/CLU. Unfortunately, formal semantics and proof rules are not mentioned for Larch/CLU, which are necessary for formal verification. CLU contains some structures, such as pointers and aliasing, that complicate formal specification and verification. Finally, CLU does not offer any solutions to the inefficiency inherent to copying. In summary, Larch/CLU has some, but not all, of the constructs necessary to encourage the design and implementation of reusable software components.

2.6.9 Z

Z [Spivey 89] is a specification language defined independently of any programming language, and is useful for formally specifying mathematical theories and operations. Mathematical theories are specified by defining a signature and a set of axioms, and it is possible to make mathematical definitions (e.g., concatenation of strings). Theories and definitions can be generic. Z is similar to the Larch Shared Language, although the syntax is quite different. A library of mathematical theories, called the mathematical toolkit, is provided, and contains definitions of useful theories such as sets, relations, functions, numbers, sequences, and multisets.

An operation is formally specified by defining its effect on a state space (i.e., one or more mathematical variables) using pre- and post-conditions. Thus, a reusable part can be formally specified by defining a state space (i.e., set of mathematical variables) that mathematically models the data, and operations that affect that state space. Unfortunately, there are no encapsulation mechanisms in Z, so the the specification of the part is simply a set of independent specifications.

Z is not associated with any programming language, but is strictly a specification language. For this reason it does not include mechanisms necessary for formal verification of programs, such as code annotations and proof rules for code. These structures need to be added to Z to make it useful for development and verification of real programs.

As discussed in Section 2.3.1, programs in a programming language not specifically designed for formal verification will most likely be difficult to specify and verify because the language probably has constructs, such as pointers and implicit aliasing, that frustrate formal specification and verification. For this reason the value of specification-only languages, such as Z, for formal verification is probably minimal.

In summary, Z is a specification language that is not associated with any programming language. It is used to formally specify mathematical theories and operations, and is powerful enough to formally specify a reusable part. Obviously, the specification of the part (written in Z) is separate from its implementation (coded in some programming language). However, because it does not include the capacity for actual code, it does not address the remainder of the reusable software part issues, such as multiple and

efficient implementations. Therefore, Z alone is not appropriate for designing and implementing reusable software components, nor is Z in combination with any implementation language we know.

2.7 Summary

Reusable software refers to software components that can be incorporated into a variety of programs *without modification* (except possibly parameterization). Furthermore, reusing these software parts should not compromise other software engineering principles such as information hiding and data abstraction. Designing and building reusable software components potentially improves both software quality and programmer productivity for three reasons — it is cost effective to commit the necessary resources to design the components properly, the designer will likely take the job seriously and design a higher quality part, and it is usually easier to reuse a well-designed component than to design and implement one on the fly. Unfortunately, there are several technical and non-technical impediments to widespread software reuse.

A reusable component has several characteristics. First, its functionality is formally specified, which serves as an unambiguous contract between a client and implementer of a part, and is essential for formal verification. Second, the specification and implementation of a part exist in separately-compilable units, which enforces the principle of information hiding, opens up the possibility of multiple implementations of a specification, and allows a client to be compiled and verified before an implementation is coded. Third, a component is generic (i.e., parameterized) if at all possible, which allows one specification to describe an infinite collection of related software components. Fourth, a component potentially has multiple implementations, each with possibly different performance, allowing a client programmer to select the implementation that best suits the application. Finally, a reusable component has efficient implementations even when it is a generic part.

There are two general approaches to encapsulation. The abstract data object approach encapsulates data with the operations that manipulate that data, and is the basic paradigm associated with “object-oriented” programming and languages. The abstract data type approach encapsulates a type with operations that have one or more parameters of that type.

There are two popular approaches to formally specifying software. In the algebraic approach, a mathematical theory is developed for each reusable component, which, in order for it to be useful, must be shown to have several properties, such as consistency, and soundness and completeness with respect to the intended interpretation. In the model-based approach, the data and operations of a part are specified in terms of existing mathematical theories, for which important properties and theorems have already been proved.

The design of a programming language has a profound impact on the ability to design and implement reusable components. For example, uncontrolled pointers and aliasing complicate the formal definition of a language as well as specification and verification of programs written in that language. Defining copying as the only data movement primitive in a language leads to designs of generic procedures that have no efficient implementation. Also, issues relating to types and variables — what are they, when are two types equivalent, etc. — affect the complexity of the language's formal language definition and the ability to write readable specifications and to formally verify programs.

A survey of several important programming languages validates the first point of the thesis — namely, that no modern programming language has all of the constructs necessary to encourage and facilitate the design of reusable software components, as defined in this chapter.