ABSTRACT

As software systems have become more complex, a search for better abstraction mechanisms has led to the use of abstract data types (ADTs). To more appropriately use ADTs, however, it is imperative that their properties and characteristics be understood. In this paper we present a method of assessing the quality of ADTs in terms of cohesion and coupling. We argue that an ADT that contains and exports only one domain and exports only operations that pertain to that domain has the best cohesive properties, and we argue that ADTs that make neither explicit nor implicit assumptions about other ADTs in the system have the best coupling properties. Formal definitions are presented for each of the cohesion and coupling characteristics discussed. Their application to Ada packages is also investigated, and we show how a tool can be developed to assess the quality of an Ada package that represents an ADT. We analyzed nearly one hundred Ada ADT packages found in Ada text books, articles about Ada, and student projects and discovered that more than half of them had inferior cohesive characteristics and almost half of them allowed inferior coupling characteristics.

Keywords: abstract data types, quality assessment, cohesion and coupling, Ada

1. INTRODUCTION

Abstraction is a powerful mechanism that helps us manage the increasing complexity of today's software systems. The predominant abstraction organizations underlying the architectures of most software systems are structured and functional programming. Structured programming constructs such as begin-end blocks, while loops, and if statements organize abstractions at a low, operational level. Functions and procedures, which are the abstractions for functional programming, organize abstractions at a higher level.

More recently, attempts to find even better abstraction-organization mechanisms have led to system architectures based on abstract data types (ADTs). An ADT is an abstraction mechanism that encapsulates a set of values together with a set of operations that apply to the values. Usually the computer representation for the set of values and the implementation of the operations are hidden from the user of the ADT.

Although an ADT-based architecture can be implemented in any language, some newer languages such as Ada, Modula II, CLU, and C++ provide syntactic mechanisms that allow a user to more easily specify and manipulate ADTs. In particular, the syntax of such languages allows a user to encapsulate an ADT and to define what information associated with an ADT is exported and what is hidden. Usually, type names and function- and procedure-invocation syntax is exported, and implementation information is hidden.

The intent of using ADTs, ADT-based architectures, and languages containing syntactic constructs that directly support ADTs is to create better-quality software -- software that is easier to understand, maintain, and reuse. As happens for any proposed mechanism for developing software, however, it is possible to misuse these ADT-based mechanisms, and the result may be poorer-quality rather than better-quality software.

What is needed, therefore, are some well-founded guidelines for developing and writing ADTs and organizing them into ADT-based architectures. Tools that support these guidelines by enforcing well-accepted rules or finding flaws and suggesting improvements are needed to assist software developers follow recommended guidelines.

As a step towards understanding how to create quality ADT-based software systems in Ada, we discuss in this paper means of assessing the quality of ADTs implemented as Ada packages. We also show how tools can be developed to provide quality assessment and suggest improvements.

Our quality assessment measures are based on cohesion and coupling characteristics. Cohesion qualitatively measures the extent to which the subcomponents of a software module are tightly bound and united throughout. Coupling qualitatively measures the independence of software modules by actual and potential connectivity.

Stevens, Myers, and Constantine have provided excellent descriptions of cohesion and coupling for function-based architectures. These authors describe cohesion by various "strength" levels. The highest strength level is functional strength, which designates a function or procedure that performs one and only one action on one entity. Myers describes an even higher level of cohesion that is equivalent to an ADT, but does not describe its effect on system architecture. Coupling also has various levels. While the best form of coupling is no coupling, this is impossible to achieve for all pairs of functions and procedures in a software system. Where coupling must exist (e.g., when one function invokes another), the best level is data coupling, which designates a form of coupling where all shared data is passed through parameters and the data for every parameter is fully used.

Cohesion and coupling have also been discussed in the context of Ada tasking, and as part of a taxonomy that suggests a classification of Ada packages. In an earlier paper, we introduced some ideas for cohesion and coupling for ADTs. In this paper we formalize, refine, and extend these ideas (Section 2). In addition, we investigate their applicability to Ada software development (Section 3). We also report the results of a case study in which we examined and classified ADTs written as Ada packages according their cohesion and coupling characteristics (Section 4). Finally, we summarize our results and make some concluding remarks (Section 5).
2. COHESION AND COUPLING CHARACTERISTICS FOR ADTS

As a working definition, we let an abstract data type (ADT) be a pair \( (D, P) \) where \( D \) is a nonempty set of exported domains and \( P \) is a nonempty set of exported operations. A domain is a set of values. An operation is a function, procedure, or constant. (A constant, of course, can be viewed as a zero-ary function that always computes the same value for the result.) Associated with an ADT are the domain and operator implementations, which are not exported.

An ADT can be mapped into an Ada package as follows. Each exported domain is represented in the visible part of a package specification by a type declaration. Each exported operation is represented in the visible part of a package specification by a function or procedure specification or by a constant declaration. Domain implementations that are not exported are contained in the private part of a package specification, and operator implementations are contained in the package body. There are some subtleties in this mapping (and especially in the inverse mapping), but we defer a discussion of these subtleties until Section 3.

Figure 1 shows an example of a package that is used in a program in which a word extracted from a text file is classified as either a noun, a verb, or neither a noun nor a verb. The exported domains are Word-type and Word. The exported operations include Create, Display, Value_of, Length_of, Same_kind, Same_word, Kind_of, Classify, and Empty_word. The implementation of Word_type, which is an enumeration, is also exported, but the implementation of Word is hidden in the private part of the package specification. Because the enumeration for Word-type is exported, several operators including the comparison operators, the attribute operators such as Succ and Pred, the membership operators In and Not In, and the assignment operator are also exported. Because Word is private (rather than limited private), the equality operator and the assignment operator for Word are also exported. The implementation of the specified operations is hidden in the package body.

Although not chosen at random, the package in Figure 1 was not created for this paper, but was selected from among those ADTs examined in the case study. The ADT was chosen because it illustrates more of the characteristics of our classification scheme than any other ADT examined. To make it suit our purposes even better, however, we did make some slight changes. We altered the formatting to make it more compact; we changed some identifier names to make them more mnemonic; and, to illustrate a point we make later, we replaced the variable declarations for Line.chars and Line_length in procedure Classify by the type declaration Line and the variable declaration Input_line.

2.1. Cohesion

Cohesion means tightly bound and united throughout. A highly-cohesive ADT should have a unified structure that represents a single object class along with operations that apply (and only apply) to objects in the object class. As such, a highly-cohesive ADT is a model of the behavior of an object in the object class represented by the domain.

Several characteristics of ADTs have an important bearing on cohesiveness. We identify and define these characteristics in terms of various cohesive strength levels in the five subsections below. The presentation order of these characteristics (generally speaking) is from weakest to strongest strength level. A particular ADT may have one or several of these characteristics.

Separable

Let \( D = \{d_1, d_2, \ldots, d_n\} \) be the exported domains of an ADT \( A \), and let \( P = \{p_1, p_2, \ldots, p_m\} \) be the exported operations of \( A \). An operator \( p_i \) is said to reference a domain \( d_j \) if \( p_i \) accepts (as an argument) an object of \( d_j \) or returns (as a value) an object of \( d_j \). A reference graph for ADT \( A \) is a directed graph \( G = (V, E) \) with vertices \( V = D \cup P \) and edges \( E = \{(p_i, d_j) \mid p_i \text{ references } d_j\} \). ADT \( A \) has separable strength if the reference graph for \( A \) is disjoint.

We name ADTs with disjoint reference graphs "separable" because they are (already) partitioned into separable subparts. The reference graph for the ADT in Figure 1 is not disjoint because the operator Kind_of references both Word and Word_type. Hence, the ADT in Figure 1 is not separable. It would be if the operator Kind_of were not present. Without Kind_of, the ADT could be split into two ADTs -- one for Word_type and its operators and one for Word and its operators.

It is possible that one of the disjoint subgraphs is degenerate -- has no edges. This happens if there is an operator that references none of the domains or if there is a domain referenced by none of the operators.

Separable-strength ADTs should be partitioned into more fundamental modules. Any operators that do not reference a domain of the ADT and any un referenced domains should be removed. Each remaining disjoint subgraph of the reference graph should be written as a separate ADT.

Multifaceted

Let \( D \) be the set of exported domains for ADT \( A \). ADT \( A \) has multifaceted strength if \( |D| \geq 2 \) and the reference graph for \( A \) is connected.

We name ADTs with more than one domain whose set of exported operators collectively applies to all domains "multifaceted" because of the multiple object classes involved. It is as if the ADT has many facets, each of which can be seen as one of the object classes. The ADT in Figure 1 is multifaceted. One "facet" is words; another is word types.

ADTs that have multifaceted strength should be disentangled. A separate ADT should be created for each domain. Each operator that references only one domain should be encapsulated with the domain it references. Other operators can be placed in the ADT of one of their referenced domains. (We further discuss the placement of operators in the next subsection.)

In disentangling an ADT, distributing its operations, and reencapsulating it as several ADTs, operator implementations may need alteration. For example, an operator may have accessed the implementation of a domain with which it is no longer encapsulated. In this case, the accessed information that is required should be exported so that it is available to operators needing the information.

In a straightforward disentanglement of the ADT in Figure 1, the only operator whose placement is not immediate is Kind_of. All other operators reference only one domain and do not access domain implementations except information that is (already) exported. For this example, Kind_of can be encapsulated without alteration in an ADT whose (only) domain is Word.

Non-delegation

To define non-delegation strength, we must consider composite domains, subcomponent-store and -retrieve operations, and computed-store and -retrieve operations.

A domain is said to be composite if domain elements have two or more subcomponents. Records, arrays, files, lists, and
package Description_words is
    type Word_type is (noun, verb, extraneous);
    type Word is private;
    procedure Create (With_this_string : in out String;
        Length_of_string : in Positive;
        The_word : out Word);
    procedure Display (The_word : in Word);
    function Value_of (The_word : in Word) return String;
    function Length_of (The_word : in Word) return Positive;
    function Same_word (Word_1, Word_2 : in Word) return Boolean;
    function Same_kind (Word_kind_1, Word_kind_2 : in Word_type;
        Word_kind_2 : in Word_type) return Boolean;
    function Kind_of (The_word : in Word) return Word_type;
    procedure Classify (The_word : in out Word);
    Empty_word : constant Word;
private
    type Word is record
        Kind : Word_type;
        Contents : String(1..20);
        Length : Positive;
    end record;
    Empty_word : constant Word :=
        (Kind => extraneous,
        Contents => (1..20 => ' '),
        Length => 20);
end Description_words;

with Responses, Text_io, Prompts, CRT_package, Line_numbers;
use Text_io;
package body Description_words is
    procedure Convert_to_lower_case(The_string : in out String;
        String_length : in Positive) is
        Lower_case_offset : constant := 32;
    begin
        for Next_character In 1..String_length loop
            if The_string(Next_character) In 'A'..'Z' then
                The_string(Next_character) := Character'Val(Character'Pos(The_string(Next_character)) + Lower_case_offset);
            end if;
        end loop;
        end Convert_to_lower_case;
    procedure Create (With_this_string : in out String;
        Length_of_string : in Positive;
        The_word : out Word) is
    begin
        Convert_to_lower_case(With_this_string, Length_of_string);
        The_word.Contents(1..Length_of_string) := With_this_string;
        The_word.Length := Length_of_string;
        end Create;
    procedure Display (The_word : in Word) is
    begin
        Put(The_word.Contents(1..The_word.Length) & " ");
        end Display;
    function Value_of (The_word : in Word) return String is
    begin
        return The_word.Contents(1..The_word.Length);
        end Value_of;
    function Length_of (The_word : in Word) return Positive is
    begin
        return The_word.Length;
        end Length_of;

Figure 1 continues to next page ...
function Same_Word(Word_1, Word_2 : in Word) return Boolean is  
begin  
return Word_1.Contents(1..Word_1.Length) = Word_2.Contents(1..Word_2.Length);  
end Same_Word;

function Same_kind(Word_kind_1 : in Word_type;  
Word_kind_2 : in Word_type) return Boolean is  
begin  
return Word_kind_1 = Word_kind_2;  
end Same_kind;

function Kind_of(The_word : in Word) return Word_type is  
begin  
return The_word.Kind;  
end Kind_of;

procedure Classify(The_word : in out Word) is  
The_prompt_kind : constant Prompts.Prompt_type := Prompts.Noun_verb;  
The_response : Responses.Response;  
Response_text : String(1..20);  
begin  
Prompts.Display(The_prompt_kind);  
Responses.Get(The_response);  
loop  
declare  
type Line is record  
Line_chars : String(1..CRT_package.Max_line_length);  
Line_length : Natural range 0..CRT_package.Max_line_length;  
end record;  
Input_line : Line;  
Wrong_response : exception;  
begin  
Response_text(1..Responses.Length_of(The_response)) := Responses.Value_of(The_response);  
if Responses.Length_of(The_response) > 0 then  
case Response_text(1) is  
when 'n'|'N' => The_word.Kind := noun; exit;  
when 'v'|'V' => The_word.Kind := verb; exit;  
when others => raise Wrong_response;  
end case;  
else The_word.Kind := extraneous; exit;  
end if;  
exception when Wrong_response =>  
CRT_package.Erase_line;  
loop  
Put("Input must be ""n", "v", or <return>; " &  
"hit <return> to continue" & ascii.be1);  
Get_line(Standard_input, Input_line.Line_chars, Input_line.Line_length);  
CRT_package.Erase_line;  
if Input_line.Line_length = 0 then exit;  
end if;  
end loop;  
Line_numbers.Display;  
Display(The_word);  
Prompts.Display(The_prompt_kind);  
Responses.Get(The_response);  
end loop;  
end Classify;

end Description_words;

Figure 1. Sample ADT Written as an Ada Package.
trees are often used to represent composite domains. Logically, there need be no upper-bound on the number of subcomponents (e.g., unbounded stacks have no logical upper-bound). There is also no restriction on the types. All the subcomponents may have the same type (e.g., homogeneous lists), or they may all have different types (e.g., heterogeneous lists).

Let $d$ be an exported, composite domain of an ADT $A$, and let $C = \{c_1, c_2, \ldots \}$ be the subcomponents of $d$. An operator $p$ of $A$ is said to be a subcomponent-retrieve operator if its input includes an object in $d$, its only result is an object in the domain of $c_i$ for some $c_i \in C$, and the only action $p$ performs is to retrieve the subcomponent. An operator $p$ of $A$ is said to be a subcomponent-store operator if its input includes an object in $d$ and an object in the domain of $c_i$ for some $c_i \in C$; its only result is an object in $d$, and the only action $p$ performs is to store the subcomponent. In Figure 1, the operators Value_of, Length_of, and Kind_of are subcomponent-retrieve operators. The other operators are neither subcomponent-retrieve nor subcomponent-store operators. Neither Create nor Empty_word includes Word as input, and there is at least one action performed in each of Display, Same_word, Same_kind, and Classify that is neither retrieve nor store.

Computed-retrieve operations arise when a logical subcomponent of an object can be computed from other subcomponents (and thus need not be stored) or can be retrieved from within other subcomponents (and thus need not be stored separately and redundantly). For example, a person's age may be computed from the person's birth date and the current date (supplied as a parameter or by the system); a person's ideal weight may be computed from the person's birth date; and a person's surname can be retrieved from within the person's full name. An operator $p$ of an ADT with domain $d$ is said to be a computed-retrieve operator if its input includes an instance of $d$ and $p$ has a single result object that is not in the domain of a subcomponent of $d$ and is obtained by retrieving subcomponents of the instance of $d$ and invoking a sequence of one or more object-generation or subcomponent-retrieve operations that collectively use a nonempty subset of the subcomponents of the instance of $d$ and all values passed into $p$. The object-generation and subcomponent-retrieve operations invoked must be operations in other ADTs (although an optimizing compiler may compile the invoke inline). For example, a person's ideal weight is obtained by retrieving the height, sex, and birth date of the given person, invoking a routine to compute the age (which should be in an appropriate ADT representing age), and invoking a routine to compute ideal weight (which should be in an appropriate ADT representing weight). A person's surname is obtained by retrieving the full name of the person and invoking a subcomponent-retrieve operation (which should be in an ADT representing people's names and either having surname as one of its subcomponents or having a computed-retrieve operator that extracts the surname from a string). None of the operators in Figure 1 is a computed-retrieve operator.

Computed-store operations are the inverse of computed-retrieve operations. A computed-store operation need not exist for every computed-retrieve operation and in some cases do not exist. An operator $p$ of an ADT with domain $d$ is said to be a computed-store operator if its input includes an instance of $d$ and an object to be stored that is not in the domain of a subcomponent of $d$ and is stored by retrieving subcomponents of the instance of $d$ and invoking a sequence of one or more object-generation or subcomponent-retrieve operations followed by subcomponent-store operations. Collectively, these operations must use a nonempty subset of the subcomponents of the instance of $d$ and all values passed into $p$. For example, given a person and a new surname, the person's full name can be retrieved and the surname can be changed. Given a person's age (to the nearest day) and the current date, the person's birth date can be changed to make the person the specified age (some women -- and men -- try to do this quite often). Since the ideal-weight computation, has no rational inverse, there should be no computed-store operator for ideal weight. None of the operators in Figure 1 is a computed-store operator.

Let $d$ be an exported, composite domain of an ADT $A$, and let $C = \{c_1, c_2, \ldots \}$ be the subcomponents of $d$. An ADT $A$ has non-delegation strength if there exists an exported operator $p$ of $A$ that references $d$, but $p$ is not a subcomponent-store or -retrieve operator, and uses only a proper subset of $C$ for all possible invocations of $p$.

We name an ADT with these properties "non-delegation" because it contains an operator that should logically be delegated to a "more-elementary" ADT with fewer subcomponents. Observe that the definition includes no mention of any more-elementary ADT; indeed, for testing non-delegation strength it is not required that the more-elementary ADT even exist.

The ADT in Figure 1 has non-delegation strength characteristics. The subcomponents of Word are the set (Kind, Contents, Length). Create is not a subcomponent-store or -retrieve operator, is not a computed-store or -retrieve operator, and uses only Contents and Length. Hence, it can be delegated to an ADT whose subcomponent set is (Contents, Length). Display and Same_word have the same characteristics as Create and should also be delegated. Classify and Empty_word use all fields of Word and should therefore not be delegated.

The reason we are interested in a subpart over all invocations is that an operator that operates on a proper subset of subcomponents in one invocation may operate on a different proper subset of subcomponents in another invocation and, over all possible invocations, may operate on all the subcomponents of an object. In a binary tree, for example, only a small subset of the nodes is accessed for a particular search, but all possible searches allow all nodes to be accessed.

Operators that have non-delegation characteristics should be removed and delegated to ADTs to which they should belong. In some cases an operator should be rewritten as two (or more) operators and placed in appropriate ADTs. It may be necessary to create new ADTs to accommodate these operators. The intent is to create ADTs in which operators either (1) extract or update a single subpart (which is either actually stored or logically computed) or (2) apply to entire objects in the domain, not just to subparts of objects.

Concealed

To define concealed strength, we must consider domains that are not exported and domains that are embedded within other domains. The domain specification, Line, in the Classify procedure in the package body of the ADT in Figure 1 is an example of a domain that is not exported. Embedded domains may either be syntactically recognizable or syntactically hidden. Declaration of a variant record and declaration of a composite domain as one of the fields of a record are examples of syntactically-recognizable embedded domains. There are two kinds of syntactically-hidden embedded domains: hidden generalizations and hidden specializations. Definitions for and examples of syntactically-hidden embedded domains are given in the next two paragraphs.

Let $d$ be an exported, composite domain of an ADT, and let $C$ be the subcomponents of $d$. Let $C'$ be a nonempty proper subset of $C$. If $C'$ constitutes a recognizable object, then $C'$ is a hidden generalization. In Figure 1 (Contents, Length) is a hidden generalization because it is a proper subset
of \{Kind, Contents, Length\} in the declaration of Word and is recognizable as a declaration for a simple word. Careful consideration of the package in Figure 1 reveals that Word is really a classified word, which contains both a simple word and a word-type classification. Simple word is a generalization of classified word and should be separately encapsulated.

In a program segment \(s\), let \(x\) be a variable declared to have type \(t\) whose associated domain of objects is \(d\). Type \(t\) is said to contain a hidden specialization if for all possible invocations of \(s\), the set of objects in \(d\) that \(x\) can represent is a nonempty proper subset of \(d\). In Figure 1, both String and Positive in procedure \(Create\) contain hidden specializations. The variable \(With\_this\_string\) can only represent strings of up to twenty symbols (a specialization of String), and the variable \(Length\_of\_string\) can only represent integers in the range one to twenty (a specialization of Positive). (Note that failure to recognize these hidden domains and deal with them properly generated a bug in the package in Figure 1: in \(Create\), strings longer than twenty symbols will not be stored and the error handling is left to the system.)

Let \(A\) be an ADT. Let \(D\) be the set of domain specifications that are not exported from \(A\), \(R\) be the set of syntactically-recognizable embedded domains in \(A\), \(G\) be the set of hidden generalizations in \(A\), and \(S\) be the set of hidden specializations in \(A\). ADT \(A\) has concealed strength if \((D \cup R \cup G \cup S) \neq \emptyset\).

We name an ADT \(A\) with these properties "concealed" because another ADT is concealed within \(A\). The set of domain specifications that are not exported from the ADT in Figure 1 is \{Line\}; the set of syntactically-recognizable embedded domains in \(A\) is empty, the set of hidden generalizations is \{simple word = \{Contents, Length\}\}, and the set of hidden specializations is \{String(1..20) in String, 1..20 in Positive\}. Thus, the ADT in Figure 1 has concealed strength characteristics.

Unexported domain specifications and embedded domains in an ADT should be removed and converted into new ADTs. Since unexported domain specifications are (already) in good form, they can be used as standalone ADTs. Syntactically-recognizable embedded domains are readily identified, but their removal may require some adjustment in operator implementations. For both hidden generalizations and hidden specializations the subset of subcomponents or the subset of objects used must be identified, and a domain specification must be created. Adjustments to operators are likely to be necessary.

**Model**

An ADT has model strength if it does not have concealed, non-delegation, multifaceted, or separable strength characteristics. The following statements can be made about model-strength ADTs.

1. A model-strength ADT \(A\) exports one and only one domain. \(A\) exports at least one domain because every ADT exports one or more domains. Since a model strength ADT \(A\) is not multifaceted, either \(A\) has less than two exported domains or the reference graph for \(A\) is disjoint. But since \(A\) is not separable, the reference graph for \(A\) is not disjoint. Thus, \(A\) exports one and only one domain.

2. If \(p\) is an exported operator of a model-strength ADT \(A\) whose exported domain is \(d\), \(p\) references \(d\), and if \(d\) is composite, \(p\) is a subcomponent-store or -retrieve operator, a computed-store or -retrieve operator, or every subcomponent of \(d\) is used in at least one possible invocation of \(p\).

By Statement 1, \(d\) must be the only exported domain. Thus, \(p\) references \(d\), if for not then the reference graph of \(A\) is disjoint, but since \(A\) is not separable, the reference graph of \(A\) is not disjoint. Since \(A\) has non-delegation strength, if \(d\) is composite, \(p\) is a subcomponent-store or -retrieve operator, a computed-store or -retrieve operator or every subcomponent of \(d\) is used in at least one possible invocation of \(p\).

3. A model-strength ADT \(A\) contains no unexported domain specifications and no embedded domains. Since \(A\) does not have concealed strength, the union of the set of unexported domain specifications, the set of syntactically-recognizable embedded domains, the set of hidden generalizations, and the set of hidden specializations is empty.

Statements 1 and 3 ensure us that a model strength ADT is concerned with one and only one domain. Statement 2 ensures us that every operation applies to the one domain and should not be delegated to an ADT whose domain has fewer subcomponents.

**2.2. Coupling**

Coupling measures the connectivity of a pair of ADTs. ADT software systems that have as few connections as possible and whose connections are as simple as possible are best. It is unlikely, of course, that any ADT software system would have no connections among ADTs. Hence, we must allow coupling, but in so doing we seek the greatest possible independence among ADTs.

In our definitions we consider an ordered pair of ADTs \((A_1, A_2)\) and state the coupling properties of \(A_1\) with respect to its use of \(A_2\). A characterization of all interconnections of \(A_1\) and \(A_2\) is expressed by giving both the coupling of \((A_1, A_2)\) and \((A_2, A_1)\).

Before proceeding, we first lay some groundwork. For our definitions we will wish to consider domains and operations of \(A_2\) that are not exported. Let \(D_2\) be the exported domains of ADT \(A_2\), and let \(D_e\) be the domains of \(A_2\) that are not exported. Similarly, let \(P_2\) be the exported operations of \(A_2\) and let \(P_e\) be the operations of \(A_2\) that are not exported.

We divide the set of exported operations \(P_2\) into two disjoint sets: \(P_e\) the set of explicitly exported operations and \(P_f\) the set of implicitly exported operations. Explicitly exported operations are those operations explicitly designated for exportation, usually by some syntactic mechanism. Implicitly exported operations are the remaining exported operations and are usually specified as a side effect of a domain declaration. Either set may be empty but not both, and their union is the set of exported operations.

For the ADT in Figure 1, \(D_2 = \{\text{Word\_type}, \text{Word}\}\), \(D_e = \{\text{Line}, \text{hidden specialization in String}, \text{the hidden specialization in Positive, the hidden generalization in Word}\}\), \(P_e = \{\text{Create, Display, Value\_of, Length\_of, Same\_kind, Same\_word, Kind\_of, Classify, Empty\_word}\}\), \(P_f = \{\text{for Word\_type, /= for Word\_type, <, <=, >, >=, >, In, Not in, Address, Base, First, Image, Last, Pos, Pred, Size, Succ, Val, Value, Width, /= for Word, /= for Word\}\}\), \(P_1 = P_e \cup P_f\), and \(P_2 = \{\text{Convert\_to\_lower\_case}\}\).

Also, for our definitions, we will wish to consider which domains and operations of \(A_2\) are used in \(A_1\). Let \(U_0 \subseteq (D_2 \cup D_e)\) be the set of domains in \(A_1\) used in \(A_2\), and let \(U_0 \subseteq (P_e \cup P_f)\) be the set of operations in \(A_1\) used in \(A_2\).
Nil

In the pair of ADTs $(A_1, A_2)$, the coupling of $A_1$ to $A_2$ is nil if $(U_1 \cup U_2) = \emptyset$.

We name the coupling "nil" because $A_1$ does not use anything in $A_2$. If the coupling of $(A_2, A_1)$ is also nil, $A_1$ and $A_2$ are independent.

Export

In the pair of ADTs $(A_1, A_2)$, the coupling of $A_1$ to $A_2$ is export if $U_2 \subset D_1$ and $U_2 \subset P_1$.

We name the coupling "export" because $A_1$ uses only the explicitly-exported domains and operations of $A_2$. The ADT in Figure 1 permits the use of implicitly-exported operations and thus does not limit other ADTs that use it to export coupling. If both $\text{Word-type}$ and Word were exported as limited-private types, the ADT would allow only export coupling.

Overt

In the pair of ADTs $(A_1, A_2)$, the coupling of $A_1$ to $A_2$ is overt if $U_2 \subset D_1$, $U_2 \subset P_1$, and $U_2 \cap P_1 \neq \emptyset$.

We name the coupling "overt" because the domain definitions are usually open, often exposing operators that are not related to the object being modeled by the ADT. The ADT in Figure 1 allows overt coupling because operations inherited from the enumerated type may be invoked on objects of type $\text{Word-type}$. Operators such as First, Pos, and Succ, among others, are inappropriate in the context of the application. Not all operators inherited from a domain definition are inappropriate, but it is impossible to syntactically determine which are appropriate and which are not.

An ADT that permits overt coupling can be improved by hiding domain definitions so that there can be no implicitly-exported operations. Although every operator inherited from the domain definition can be rewritten as an explicitly-exported operator, it is only necessary to create specifications and implementations for those that are actually used, which is often a small subset of those implicitly exported. For the ADT in Figure 1, for example, there are twenty implicitly exported operators for $\text{Word-type}$, but only three $(=, =, \text{Image})$ would ever likely be used.

Some may argue that overt coupling is not bad. It is not the worst flaw, but export coupling is better. When the implementation of a type is exported, there are (almost) always several inappropriate operations also exported. If an array implementation of a stack is exported, for example, users can change any part of the stack by assignment to a subpart of the array. Most designers do not wish to provide such capabilities, but do so inadvertently when they export a type definition. Furthermore, exporting type declarations also causes modification problems. For instance, assume that a record representation for some domain is exported from an ADT and that access to the subparts is therefore provided by the "$.$" operator. If the representation is ever changed, to a linked list for example, then all "$.$" operations will need modification. These kinds of modification problems can be eliminated by avoiding overt coupling.

Covert

In the pair of ADTs $(A_1, A_2)$, the coupling of $A_1$ to $A_2$ is covert if $U_2 \cap D_1 \neq \emptyset$ or $U_2 \cap P_1 \neq \emptyset$.

We name the coupling "covert" because $A_1$ uses a domain or operation hidden in the implementation of $A_2$. If the ADT in Figure 1 were implemented without the package syntax, the operation Convert_to_lower_case would be exposed and could be invoked by a using ADT, and thus, there would be covert coupling. In Ada and other languages that allow programmers to syntactically designate which domains and operators are to be exported and that prohibit access to any domains and operators not exported, covert coupling can, to a large extent, be prevented. When using these languages, covert coupling can occur, for example, when unchecked conversion is used to gain unauthorized access to an object.

Surreptitious

In the pair of ADTs $(A_1, A_2)$, the coupling of $A_1$ to $A_2$ is surreptitious if $A_1$ uses information in $A_2$, but does not obtain the information through the use of elements in $U_p$ or in $U_y$.

We name the coupling "surreptitious" because the information used is obtained by stealth. A programmer knows something that should not be known about the implementation of an ADT and makes use of the knowledge to implement some other ADT. An example is seen in the Convert_to_lower_case procedure in Figure 1 where the programmer makes use of the fact that in the ASCII character set encoding the code for an upper-case letter can be obtained by adding 32 to the code for its lower-case. Thus, the ADT in Figure 1 uses information in the ASCII ADT, but does not obtain it from a domain declaration nor through the use of an operation.

Surreptitious coupling should be eliminated. It is, however, impossible to automatically detect with certainty. Even signs that surreptitious coupling is present may be difficult to recognize. In Figure 1, for example, a signal that surreptitious coupling is present in the Convert_to_lower_case procedure is that the plus operator, which applies to integers, is being applied to a position in a collating sequence.

3. QUALITY ASSESSMENT OF ADTS IN ADA

Assessment of software quality can best be done by automated tools. We have designed a tool to assess the coupling and cohesion properties of an ADT represented by an Ada package.* In this section we first discuss how ADTs map to and from Ada packages. We then explain how cohesion and coupling properties can be determined.

3.1. Mapping ADTs to Ada Packages

Every ADT can be represented by an Ada package.* An ADT is composed of two parts: the definition and the implementation. The implementation contains the implementations for the domain and operation definitions. An Ada package consists of two parts: the package specification and the package body. The package specification consists of two parts: the visible section and the private section. The visible section can contain variables, types (limited private, private, or visible), subtypes, function/procedure specifications, constants, exceptions, task specifications, and internal package specifications. Interface specifications of these Ada constructs are exported, and thus other program segments that import the package may use them. The private section can contain the same Ada constructs, but their interface specifications are not exported and thus cannot be used outside the package. The package body can also contain these same Ada constructs plus function/procedure implementations and task implementations. The interface specifications of constructs in a package body are not exported.

Let $n$ be a name for a domain definition $d$ exported by an ADT. We can represent $d$ in the visible part of an Ada package by

\[ \text{type } n \text{ is private} \]

* ADTs can also be represented in Ada by tasks and by unencapsulated groups of function, procedure, and type definitions, but consideration of these representations is beyond the scope of this paper.
or, if we do not wish to export the equality and the assignment operator, by

\[ \text{type n is limited private} \]

Let \( p \) be an operator exported by an ADT. We can represent \( p \) in the visible part of an Ada package by a constant specification (if \( p \) is a zero-ary operator that returns a result), by a function specification (if \( p \) has arguments and returns a result), or otherwise by a procedure specification. The implementation of a domain can be placed in the private section of a package specification or in the package body as required and allowed by Ada, and the implementation of an operator is placed in the package body.

If there is only one domain \( d \) for an ADT \( A \) and each operator applies to \( d \) and should not be delegated, observe that it is possible to create a package that has model cohesion and allows only export coupling. Domain \( d \) can be declared as a limited-private type, and each operation can be specified as an operator on \( d \). With these characteristics, \( A \) cannot have separable, multifaceted, or non-delegation strength nor overt coupling and need not have concealed strength nor covert or surreptitious coupling.

### 3.2. Mapping Ada Packages to ADTs

While all ADTs can be mapped to packages, all Ada packages are not ADTs. Various formal and informal classifications of Ada packages have been suggested.\(^{13,14,11}\)

We use the following conditions to determine if a package can be considered to represent an ADT. If a package satisfies these conditions, it is said to be an ADT package.

1. The visible section of the package specification may include only type and subtype declarations, function/procedure specifications, constants, and exceptions.
2. The visible section of the package specification must include the reserved word type or subtype.
3. If the visible section of the package specification includes no subtype declaration and all the type declarations are limited private, then the visible section of the package specification must include at least one function/procedure specification or one constant declaration.

Condition 1 excludes packages that are (probably) not intended to be ADTs and whose analysis is beyond the scope of this paper. Condition 2 guarantees the existence of at least one exported domain. Condition 3 guarantees the existence of at least one exported operator. If there is a subtype declaration, the inherited operators are implicitly exported. If there is a type declaration that is not limited private, the equality and assignment operators are implicitly exported. When there are no implicitly exported operators, one must be explicitly exported.

### 3.3. Assessing Cohesion and Coupling Properties of ADT Packages

To analyze Ada packages, we must parse the source code and create an intermediate form more suitable for analysis. This part of the analysis is not described here since it is based on classic parsing techniques using the Ada grammar.

To apply the definitions for cohesion and coupling stated above, we must create a correspondence between the defined entities in the definitions and the syntax of an ADT package. Once this correspondence is established, the definitions can be directly applied to determine the cohesion and coupling properties. This correspondence is mostly straightforward, but there are some subtleties as discussed below.

### Domain Definitions

Let \( D \) be the set of names of all type and subtype declarations in both the package specification and the package body. Note that there may be more declarations than names because some declarations may be incomplete and then completed later, using the name a second time. (Recognition of embedded domains is subtle and is discussed below.) \( D \) is the set of domains for the ADT package. The subset of \( D \) for which there exists a declaration in the visible part of the package specification is the set of exported domains. The set of non-exported domains consists of the constant that remains when the exported domains are removed from \( D \). A domain \( d \in D \) is composite if it includes an array or record specification, or an access to an array or record specification.

### Operator Definitions

The set of explicitly-exported operators is the set of function specifications, procedure specifications, and constant declarations in the visible section of a package specification. The set of implicitly-exported operators is the set of all operators, not in the set of explicitly-exported operators, that can be applied to objects in the set of exported domains. These implicit operators include the equality and assignment operators that apply when a type definition is not limited private, the operations inherited in a type definition when a type definition is neither private nor limited private, and the operations inherited in a subtype definition. The set of exported operators is the union of the set of explicitly- and implicitly-exported operators. The set of non-exported operators is the set of all function/procedure specifications and constant declarations in the private section of a package specification and in the package body plus the set of all operations inherited from non-exported domain definitions.

### Reference Definitions

Let \( d \) be a domain with name \( n \). Let an operator \( p \) be a function or procedure specification or a constant declaration. Operator \( p \) references \( d \) if \( n \) is a type name for a formal parameter of function or procedure specification \( p \), is a type name for the return type of function specification \( p \), or is the type name for constant declaration \( p \), or \( n \) is the type name of the type of constant assigned in a constant declaration \( p \). An operator \( p \) uses a domain \( d \) with name \( n \) if the implementation of \( p \) contains \( n \) and the name \( n \) refers to \( d \) (\( n \) has not been redefined) or if the operation is inherited from the definition of \( d \). An operator \( p \) uses another operator \( q \) if the implementation of \( p \) invokes \( q \).

### Subtleties

There are some subtle problems in applying the definitions. It is difficult to achieve precise definitions leading to fully automatic detection algorithms for some of our concepts such as "retrieve action", "store action", "object-generation operation", "embedded domain", and "surreptitious access". As discussed above, detection of surreptitious coupling is likely to be impossible. However, reasonably good heuristics can be developed to overcome most of the other difficulties.

There are also difficulties caused by language representation mechanisms. For example, is String(l..20) in Figure 1 a composite domain? If so, we should declare that Word in Figure 1 has a syntactically-recognizable embedded domain and split String(l..20) apart from Length as well as Kind. Recognizing String(l..20) as a generic-like construct of special design in Ada, however, makes it possible to classify String(l..20) as non-composite. Another example is the difficulty of eliminating variant records to remove concealed-strength characteristics. Since Ada lacks support for generalization, it is awkward to create aggregates of specializations without variant records, for instance to make a list of employees who are variously represented as managers, secretaries, engineers, and clerks all
having slightly different sets of attributes. Yet another example is the representation of recursive types. Since examination of the recursive-type problem leads to some interesting insight, we more fully discuss it below.

Consider the recursive type definition in Figure 2, which should normally be found in the private section of an Ada package specification with "BtypeR List Bis limited privateR" in the visible part. Our simple definitions from above would indicate that the type List_node is part of a concealed ADT. Normally, type declarations that appear only in the implementation should be removed and made into stand-alone ADTs, but this is impossible with type declarations that are part of recursive type definitions. The problem arises when trying to define a single, exported, recursive domain using syntax that requires several type definitions to express the recursive relations. To prevent an Ada analyzer from concluding that the implementation of a single recursive domain exhibits concealed cohesion, recursive type definitions should be analyzed differently.

```
type List_node;
type List is access List_node;
type List_node is record
  Item: Value;
  Next-node: List;
end record;
```

Figure 2. A Recursive Type Definition.

Recursive type definitions can be automatically recognized as follows. Let \( D \) be the set of domains of an ADT \( A \), and let \( N \) be the set of names of the domains in \( D \). Let \( r \) be the relation over \( N \) defined as \( \{(n_1, n_2) \mid n_1 \in N, n_2 \in N, \text{ and } n_1 \text{ includes } n_2 \text{ in its definition}\} \). If the directed graph of \( r \) is cyclic, \( A \) contains a recursive type definition. It may contain more than one. If the graph of \( r \) contains exactly one cycle including all nodes in \( N \), then we conclude that \( r \) represents a single recursive domain. Otherwise, \( A \) contains a concealed ADT and should be disentangled.

Although (complete) recursive definitions can also appear both in the visible part of a package specification and in the package body, they should not. A recursive definition completely specified in the visible part is (correctly) classified as having multifaceted and overt characteristics because it exports both the component part of a domain and the recursive structure. A recursive definition completely specified in the package body is classified as being concealed. Since it is a recursive domain, which cannot be split, it should be extracted as a unit and used to create a new ADT.

### Table 1.

<table>
<thead>
<tr>
<th>Group</th>
<th>Separable</th>
<th>Multifaceted</th>
<th>Non-delegation</th>
<th>Concealed</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Students</td>
<td>10%</td>
<td>20%</td>
<td>10%</td>
<td>29%</td>
<td>43%</td>
</tr>
<tr>
<td>Authors</td>
<td>15%</td>
<td>21%</td>
<td>5%</td>
<td>35%</td>
<td>28%</td>
</tr>
<tr>
<td>Authors Before 1986</td>
<td>23%</td>
<td>20%</td>
<td>0%</td>
<td>33%</td>
<td>33%</td>
</tr>
<tr>
<td>Authors 1986 and After</td>
<td>16%</td>
<td>21%</td>
<td>0%</td>
<td>33%</td>
<td>42%</td>
</tr>
<tr>
<td>Small/Standard ADTs</td>
<td>9%</td>
<td>20%</td>
<td>2%</td>
<td>32%</td>
<td>45%</td>
</tr>
<tr>
<td>Large/Original ADTs</td>
<td>24%</td>
<td>24%</td>
<td>3%</td>
<td>32%</td>
<td>34%</td>
</tr>
<tr>
<td>Students Taught Ada</td>
<td>13%</td>
<td>17%</td>
<td>7%</td>
<td>27%</td>
<td>53%</td>
</tr>
<tr>
<td>Student Who Learned Alone</td>
<td>12%</td>
<td>50%</td>
<td>17%</td>
<td>67%</td>
<td>17%</td>
</tr>
<tr>
<td>All ADTs Combined</td>
<td>17%</td>
<td>20%</td>
<td>2%</td>
<td>32%</td>
<td>39%</td>
</tr>
</tbody>
</table>

### Table 2.

<table>
<thead>
<tr>
<th>Group</th>
<th>Possible Overt Coupling (Domain Definition Exported)</th>
<th>Possible Overt Coupling (Private Rather Than Limited Private)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Students</td>
<td>38%</td>
<td>48%</td>
</tr>
<tr>
<td>Authors</td>
<td>48%</td>
<td>35%</td>
</tr>
<tr>
<td>Authors Before 1986</td>
<td>43%</td>
<td>30%</td>
</tr>
<tr>
<td>Authors 1986 and After</td>
<td>51%</td>
<td>30%</td>
</tr>
<tr>
<td>Small/Standard ADTs</td>
<td>34%</td>
<td>48%</td>
</tr>
<tr>
<td>Large/Original ADTs</td>
<td>61%</td>
<td>20%</td>
</tr>
<tr>
<td>Students Taught Ada</td>
<td>27%</td>
<td>60%</td>
</tr>
<tr>
<td>Student Who Learned Alone</td>
<td>47%</td>
<td>47%</td>
</tr>
<tr>
<td>All ADTs Combined</td>
<td>46%</td>
<td>30%</td>
</tr>
</tbody>
</table>

* The percentages across a row do not add to 100% because a single ADT can have several characteristics.

### 4. AN ANALYSIS OF ADT PACKAGES

To investigate the usefulness and applicability of our cohesion and coupling characteristics, we analyzed ninety-four Ada ADT packages. The ADTs were not chosen at random, but were taken from all available resources, seventy-three from books on Ada,13,16,18, twenty-one from Ada Letters (1982 through 1987), fifteen from source code written by senior students who had been taught Ada programming, and six from source code written by a student who had learned Ada on his own. The results of the analysis are shown in Tables 1 and 2.

Because the analyzed sample was not chosen at random, we do not provide statistical results. As we analyzed the code, however, we did make some interesting observations. It appears that people create model ADTs less than half the time (only 39% in our study). It does not seem to matter whether the programmers are authors experienced in Ada or students just learning to use the language.

The quality of ADT packages seems to be about the same both before 1986 and during and after 1986. We thought there might be some improvement, but the data does not seem to support this conclusion. The quality of ADTs seems to be marginally affected by the type of object class created. Those programmers encapsulating small or standard data structures, such as queues, complex numbers, or stacks, created packages that were a little better than those creating large or original ADTs. These small/standard packages were especially better in preventing the possibility of overt coupling. Although no
conclusions should be drawn, it was interesting to compare stu-
dents who had been taught Ada to the one student who learned
Ada on his own. The student who learned on his own pro-
duced consistently poorer packages.

One confusing problem appears to be the use of packages
to represent several types of abstraction such as abstract data
types, abstract objects, and general aggregations. Not fully
comprehending the differences between the uses of packages,
people often mix their abstractions and thus produce ADTs of
separable and multifaceted strength.

Another problem seems to be lack of education. As in the
1960's when many did not understand good functional abstrac-
tion, programmers now do not understand good ADT abstrac-
tion. This problem is exacerbated by most language's textual
representation. The textual representation makes it difficult to
observe both lower-level component abstractions and higher-
level composition abstractions. Thus, many mix both lower-
level and higher-level abstractions, resulting in inferior ADTs.
If the textual presentation could add another dimension that
would allow programmers to "zoom" into and out of abstrac-
tions, the need to localize both low-level and high-level abstrac-
tions should be ameliorated.

Still another problem seems to be the amount of typing
required to represent a small or trivial abstraction or to hide
the details of domain definitions and export only the operations
needed. In many instances it requires considerably more typing
to represent an abstraction as an Ada package than to mix the
lower-level type declarations and operator implementations
with the higher-level abstraction. This mixing usually pro-
duces separable-, multifaceted-, non-delegation-, or concealed-
strength ADTs or ADTs with possible overt coupling. Thus,
modifiability and understandability are often sacrificed for a
few minutes saved during software development.

5. SUMMARY AND CONCLUSIONS

We have presented a means of assessing the quality of
ADTs written as Ada packages. Quality assessment is based
on cohesion and coupling properties, which respectively measure
how unified and independent an ADT package is.

We classified the cohesiveness of ADTs as having separ-
able, multifaceted, non-delegation, concealed, or model
strength. We argued that a model-strength ADT is best and
that an ADT should thus logically export one and only one
domain (i.e., should not have either separable or multifaceted
strength), should logically export only operations that operate
on the one domain exported (i.e., should not have non-
delegation strength), and should contain no hidden ADTs
within the implementation (i.e., should not have concealed
strength).

We classified ADT coupling as nil, export, overt, covert,
and surreptitious. We argued that an ADT should neither
indirectly nor directly access the implementation of another
ADT (i.e., should have neither surreptitious nor covert coupl-
ing), that an ADT should not implicitly export operators
(i.e., should not allow overt coupling), but should access
another ADT only through explicitly exported operators, if at
all (i.e., should have export coupling when nil coupling is not
feasible). Thus, the only interconnection among ADTs should
be through export coupling.

We explained how ADTs can be mapped into Ada and
how a tool could be created to assess the quality of ADTs
already written as Ada packages. We observed that all ADTs
with one domain can be represented as Ada packages such that
they have model cohesive strength and allow only export cou-
pling. For already-written ADT packages, we showed how
each of the inferior cohesive characteristics as well as overt-
coupling can be detected. Several subtle problems make the
development of an automated-assessment tool nontrivial.

In a case study, we analyzed ninety-four ADT packages.
Twenty-one were written by senior-level computer science stu-
dents, and seventy-three were found in Ada text books and
articles. More than half of the ADTs analyzed had inferior
cohesive characteristics, and about half allowed inferior cou-
pling properties. Although additional effort is often required to
design and create ADT packages with model cohesive strength
and export-only coupling, we believe that the potential gain in
understandability, modifiability, maintainability, and reusabil-
ity make the effort worthwhile.

References

1. G. Booch, "Object-oriented development," IEEE Trans-
211-221, February 1986.

2. B. Meyer, "Reusability: the case for object-oriented
design," IEEE Software, vol. 4, no. 2, pp. 50-64, March
1987.

data types," Proceedings of the ACM Symposium on
Very High Level Languages, SIGPLAN Notices, vol. 9,
no. 4, pp. 50-59, ACM, New York, New York, April,
1974.

4. J. A. Goguen, J. W. Thatcher, and E. G. Wagner, "An ini-
tial algebra approach to the specification, correctness,
and implementation of abstract data types," in Current
80-149, Prentice-Hall, Inc., Englewood Cliffs, New Jer-

5. M. Page-Jones, The Practical Guide to Structured Sys-

6. B. H. Liskov and J. Guttag, Abstraction and Specifi-
cation in Program Development, The MIT Press, Cam-
bridge, Massachusetts, 1986.

7. G. J. Myers, Composite/Structured Design, Van Nostrand


9. E. Yourdon and L. L. Constantine, Structured Design:
Fundamentals of a Discipline of Computer Program and
Systems Design, Prentice-Hall, Inc., Englewood Cliffs,

10. K. W. Nielsen, "Task coupling and cohesion in Ada,"

vol. 6, no. 4, pp. 53-65, July/August 1986.

12. D. W. Embley and S. N. Woodfield, "Cohesion and cou-
lping for abstract data types," Proceedings of the 1987
Phoenix Conference on Computers and Communications,

13. G. Booch, Software Components with Ada, The
Benjamin/Cummings Publishing Company, Inc., Menlo
Park, California, 1987.

14. G. Russell, "Experiences implementing a reusable data
structure component taxonomy," Proceedings of the
Fourth Washington Ada Symposium/Fifth National
Conference on Ada Technology, pp. 8-18, Washington
DC, March 1987.

15. G. Booch, Software Engineering with Ada, Second Edi-
tion, Benjamin/Cummings Publishing Company, Menlo
Park, California, 1987.

16. N. Gehani, Ada An Advanced Introduction, Prentice-Hall,