CS323 Lecture: Data Abstraction and Object-Oriented Languages; Ada Case Study

2/4/09

Materials: Handout of Ada Demonstration Programs + Executable form (first_example, ascii_codes, rationals, stack, exception_demo, oo_demo)

I. Introduction

A. As we have discussed previously, the earliest programming language paradigm (to which many languages belong) is the imperative or procedural paradigm. However, within this paradigm there have been some important developments which have led to modern languages being significantly better than older members of the same family, as well as to the development of the object-oriented paradigm.

1. One such development is "structured programming" which focussed on the control flow of programs. A major outcome of this work has been the presence, in most modern languages regardless of paradign, of certain well-defined control structures (conditional; definite and indefinite iteration).

2. A second major development has been the evolution of the idea of DATA ABSTRACTION.

a. In the early days of computer science, most attention was given to algorithms and to language features to support them - e.g. procedural abstraction and control structures.

b. In the late 1960's, Donald Knuth published a three volume work called "The Art of Computer Programming" which focussed attention on the fact that data structures are an equally important part of software design.

i. His book is really responsible for the birth of the idea that a "Data Structures" course should be part of the CS curriculum.

ii. Historically, coverage of data structures migrated from being an upper-level course in the CS major (300 level) to being a core part of the introductory sequence.

c. The fact that algorithms and data structures are of equal importance is perhaps best captured by the title of a very influential book by Niklaus Wirth: Algorithms + Data Structures = Programs.

d. Early work on data structures led to the development of the related notion of DATA ABSTRACTION - the idea that a major component of the software design process should be the identification of key ABSTRACT DATA TYPES (ADT's). In some sense, object orientation can be understood as the ultimate outgrowth of this work.

e. So what is an ADT?

Key ideas:

i. As you recall from our discussion of data types, one way of defining a data type is as a set of values and as set of operations. An ADT is simply a programmer-defined (rather than builtin) data type that has these components.

ii. Separation of the INTERFACE (operations) from the IMPLEMENTATION. With an ADT, there is a conceptual (if not language-enforced) "wall of abstraction" that separates the two.

f. The OO paradigm takes this idea to the limit

i. In the imperative paradigm, procedures operate on data.

ii. In the OO paradigm, objects (data) operate on each other.

iii. What, in OO, corresponds to the notion of an ADT?

ASK

- It's tempting to say "the class"

- In reality, it's the interface. (But access control mechanisms in a class can be used to create a single entity that both defines and implements an interface, while preserving the wall of abstraction through the notion of private fields and methods.

3. As the idea of ADT's evolved, another key idea that emerged was ENCAPSULATION/INFORMATION HIDING - that each ADT should encapsulate its implementaton details in such a way that clients of the ADT cannot\ be written in such a way as to depend on them, thus facilitating the writing of modular software in which changes to one module do not impact other modules.

4. A final key idea is the idea of PARAMETERIZED ABSTRACT TYPES - e.g. data types that serve as generic containers that can be instantiated to serve as containers for objects of a particular type.

a. We will look at an Ada example of this shortly

b. OO has the same thing - e.g. C++ templates or Java generics

B. The programming languages that were developed at the time these ideas were developing tend to reflect the evolving state of the art.

1. For example, Pascal supported ADT's by allowing user-defined types (via the type statement) and operations on them implemented via functions and procedures. However, it did not really support information hiding - a client of a module is not prohibited by the compiler from penetrating the wall of data abstraction - something I always had to get on when grading projects in Gordon's Pascal era. (Example: my "beeping" in the middle of class)

Also, Pascal had no support for parameterized types (generics).

2. CLU - developed at MIT in the 1970's - was the first language to incorporate full support for data abstraction, but it has never become widely used.

3. Modula-2 (Niklaus Wirth's successor to Pascal) added two new concepts to support information hiding.

   a. A Modula-2 program consists of a collection of MODULE's, each of which consists of two separately compiled parts - a DEFINITION MODULE and an IMPLEMENTATION MODULE.

      i. Since modules are separately compiled, in general they cannot depend on each other. The information in one compilation unit is hidden from all others.

      ii. There is one exception: the compilation of a definition module produces a symbol file, which is read by the compiler when it is compiling the corresponding implementation module, and any other module that explicitly IMPORTs it. Thus modules that ask to do so can utilize information contained in the definition part of another module; but cannot know about anything in the implementation part of another module.

   b. Modula-2 supported information hiding by introducing the concept of an OPAQUE TYPE - a type which is partially declared in the definition module and fully declared in the implementation module. Other modules can only depend on the information about the type contained in the definition module.

   Modula-2, however, had no support for parameterized types.

4. The language we will be using as a case study - Ada - represents the first widely used language to fully support data abstraction, including parameterized types. We will learn more about this when we turn to the language.

C. As we have said, the idea of data abstraction finds further development through object orientation, a new paradigm which essentially turns the relationship between algorithms and data around.

1. Early imperative language were based on software design methodologies that focussed on algorithms (e.g. top-down design), with data tructures subordinate to them.

2. Data abstraction languages give algorithms and data structures coordinate roles.

3. OO languages focus on the data objects of a program, with algorithms encapsulated within the objects as methods.

   Note: C++ is in some sense both a data abstraction language and an OO language, as is Ada 95.

D. We turn now to Ada - a language which started out as a data abstraction language, but has since evolved (in Ada 95) to have OO characteristics as well.

II. The History and Characteristics of Ada

A. Ada was born out of a software crisis in the Department of Defense (DOD) that became apparent in the 1970's.

   1. Dozens of higher-order languages in use + much code written in assembly language.

   2. Software development costs were growing in proportion to the SQUARE of program size.

   3. In the mid 70's, DOD's software budget was $3 billion / year

B. To address this crisis, the DOD decided to design a new, single language that could address some of the perceived problems.

   1. Major design goals

      a. Facilities for modularizing using Abstract Data Types, information hiding, etc. (then new concepts in software engineering). This was intended to make development cost grow only linearly with program size.

      b. Machine-independence.

      c. One language for all purposes (DOD decree: thou shalt use this)

      d. A language suitable for use in embedded systems.

      e. Capabilities for concurrent processing.

   2. The design process began with a series of increasingly refined specifications:

      a. Strawman - 1975
      b. Woodenman - 1975
      c. Tinman - 1976
      d. Ironman - 1978
      e. Steelman - 1979

   3. The actual design was done competitively - eventually narrowed to four teams (denoted by colors.) The competition was won by Team Green, headed by Jean Ichbiah of Honeywell Bull.

C. The language was named Ada in 1979, in honor of Ada Augusta, Countess Lovelace, history's first computer programmer (on Babbage's Analytical Engine.)

   1. The final description was published in 1980.

   2. The name Ada is trademarked by the DOD. DOD does not allow any subsets or supersets of Ada, and uses its trademark to prevent anyone from calling such a language Ada.

3. In fact, no implementor is allowed to call a compiler Ada until it has passed a DOD validation suite of tests that confirm that the language it compiles is exactly the official Ada language - no more and no less.

   In the literature, one will often see a notation like Ada *, with a footnote reading:

   * Ada is a trademark of the US Department of Defense

4. For a number of years, it was madated that any mission-critical software used by the DOD had to be written in Ada. Though this mandate has been rescinded, Ada is still the preferred language for DOD applications and for many other life-critical areas (such as avionics).

   As one proponent of Ada put it "Software used to kill people and break things MUST be reliable". (http://www.drew-hamilton.com/stc99/stcAda_99.pdf)

5. The original Ada specification - now known as Ada83 - was modified to add improved support for OO to become Ada95, and there was a more recent (but minor) modification in 2005. All versions of Ada are also ISO standards, and the Ada 95 Language Reference Manual is freely available at http://www.adaic.org/standards/95lrm/html/RM-TTL.html. (Also linked as a Course Document from Blackboard site).

   Ada 95 is downward compatible with Ada 83 - an Ada 83 program is, in most cases, also a valid Ada 95 program. (There are a very few exceptions to this statement, mostly involving constructs a programmer is unlikely to use, and most of which will be detected by the compiler so the programmer will know to fix them.) We will focus first on the features common to both dialects, then on the new features of Ada 95.

6. Probably the most widely used Ada compiler is gnat - the Gnu Ada Translator - whose development was partially funded by the U.S. Air Force. (It is, like all Gnu software, free software.)

D. Ada is in the Algol tradition, and is - in particular - a descendant of Pascal.

E. first_example.adb - HANDOUT, RUN

   Observations:

   1. Like Pascal, Ada is not case-sensitive, but there is a preferred style:

      a. Use of lower-case letters for reserved words.

      b. Case conventions for identifiers:

         Ada83: all upper-case letters preferred for identifiers.

         Ex: INTEGER (considered an identifier in Ada)
             TEXT_IO (the standard text IO package)
             CONSTRAINT_ERROR (a predefined exception)

Ada95: Identifier names begin with an upper-case letter, but remaining letters are lowercase. Multiple words use a "_" between words, plus mixed case.

Ex: Integer
    Text_IO
    Constraint_Error

c. gnat requires that a compilation unit be stored in a file that has the same name as the compilation unit, but all lowercase, with a file type of either .adb or .ads (we will discuss what each means shortly). Gnat also limits filenames to 8 characters in length - longer compilation unit names are "krunched".

2. Comments are line oriented, and begin with --. Everything following the -- on the line is taken as comment.

3. Use of "Noise words" (sometimes instead of symbols) - e.g.

   procedure First_Example is

4. No I/O facilities are defined as part of the language. All I/O operations are done through use of routines contained in library packages (as in C).

   (This program uses Text_IO, which manages input-output of strings. There are other packages for handling IO of numbers.)

   a. with Text_IO is a context clause that specifies that the package Text_IO is needed for the compilation.

   b. use Text_IO allows reference to the elements of Text_IO without a qualifier. If this were omitted, one would need to say Text_IO.Put_Line("Hellow, world!");

5. Use of semicolon as a statement TERMINATOR (as in C), not a statement SEPARATOR (as in Pascal).

6. An end statement specifies what is ending - here the procedure First_Example

7. Many of these are departures from the Pascal tradition that were motivated by studies which showed that they lead to more reliable programs.

F. ascii_codes.adb - RUN

   Observations

   1. While IO for Strings is a regular package, IO for integers is a generic package that must be instantiated for different integer types due to Ada's strong typing. Note that the instantiation and use occur inside the procedure - not as a context clause).

   2. Ada doesn't support variable length strings, per se - instead, one declares a String of suitable maximum length and then keeps track of the actual length separately. Get_Line in Text_IO has a variant that gives the actual length of a line read.

3. In keeping with the block-structured heritage of the language, it is possible for a subprogram to contain local procedures or functions, not visible outside the subprogram body.

4. Ada distinguishes between procedures - which return no value - and functions which return a value. Notice that a function definition specifies the return type - a procedure does not need to. (Compare void in C/C++/Java).

5. Ada has for and while loops, with syntax

```
while Condition loop
for Variable in range loop
```

Either type of loop is ended by end loop

6. The ordinal value (ASCII code) for a character is obtained by the function Character'Pos applied to the character. There is a similar function Character'Val which converts an integer to a Character.

7. Any begin .. end block can have an exception handler attached. Format is exception, followed by clauses for the various kinds of exceptions and what is to be done. (Compare Java try ... catch(<exception type>)

8. End-of-file on input raises an End_Error exception; this can be handled to produce nice program termination.

III. Ada Compilation Units

A. An Ada program consists of one or more compilation units, each of which is either a subprogram or a package or a task or subunit of another compilation unit. Subprograms and packages may, in turn, be split into separate specification and body units.

1. A subprogram is either a function or procedure, which may be the main program of the entire program, or may be a function or procedure that is to be called by other compilation units.

   As in any block-structured language, the body of the function or procedure may contain definitions of local variables, procedures etc. However, only the outermost function/procedure declaration is available to other compilation units.

   Example: In ascii_codes, the variables Line and Length and the function Code nested inside Ascii_Codes are strictly local to that procedure

2. A package is a named collection of type, object, and function/procedure declarations. Each of the "top-level" declarations is available to other units.

3. We will discuss tasks later

4. A compilation unit can consist of two parts: a specification and a body. These may appear together in the same source file, or they may be in separate files. The specification provides information available to other compilation units; the body implements the specification.

Example: rationals.ads/.adb, ratdemo - RUN ratdemo

Observations:

[rationals.ads]

a. The package rationals defines an abstract data type Rational, with the basic arithmetic operations plus Get and Put for IO.

   i. In the specification, the representation for a rational number is made private, which means that, while a client program may declare variables of type rational, it cannot access the actual implementation (i.e. cannot refer to Numerator and Denominator).

      Why does the specification need to include the complete private declaration for the type Rational?

      ASK

      So the compiler knows how much space to allocate for a Rational declared in a client unit.

   ii. In the specification, the various operations on Rationals are declared - but the declaration ends with a ";" rather than "is" followed by a body.

b. It is possible to overload the standard arithmetic operators for ADT's. (Note, too, that "-" is overloaded twice - once as a binary operator and once as a unary).

[rationals.adb]

c. The package body includes three functions/procedures not declared in the specification. Which ones?

   ASK

   These are only for use by the implementation - a client cannot use them. (It is also possible for a subprogram or package body to contain local packages, which are not visible outside the subprogram or package body.)

d. Procedure/function headers are repeated in the body.

e. Format of an if statement in Greatest_Common_Divisor. Variants:

```
if .. then
    ...
end if;

if .. then
    ...
else
    ...
end if;
```

```
if .. then
    ...
elsif ... then
    ...
elsif ... then
    ...
else
    ...
end if;
```

[Note difference between elsif and else if - latter requires another end if]

f. Parameter modes - note heading of Reduce

   i. in parameters - function/procedure may use, but may not change

   ii. out parameters - procedure may change (assign to), but may not use.

   iii. in out parameters - both of the above

   iv. If a parameter mode is not specified, in is assumed. (Some of my examples explicitly specify in, others leave it implied, to show both approaches. There is no semantic difference).

   v. Functions can only have in parameters - not out or in out

g. Mid-exit from loop in body of Get - exit when Condition

[ratdemo.adb]

h. The context clauses of a client can refer to user-written packages as well as library packages. The client can refer to things declared in the user-written package - e.g.

   i. The variable declarations of type Rational

   ii. All of the calls to Get and Put invoke the methods declared in Rational because of the type of the parameters

B. A complete Ada program will oftenn be split over multiple source files, each of which contains the specification or body of a compilation unit. However, compilation is initiated by compiling the unit which contains the main program, which the causes other units to be compiled as well.

DEMO: Delete all files on except the three sources, then

    gnat make ratdemo.adb

    Show the resultant files

C. One important feature of Ada is the ability to declare GENERIC compilation units. (This facility is similar to, but by no means identical with, the template facility provided by C++ or the generics supported by Java 5 and later).

1. A generic unit begins with the word generic, followed by parameter specifications, followed by a subprogram or package specification.

2. A generic unit stands for a family of units, each of which differs from the others in certain parameters.

3. A new instance of a generic package can be instantiated by giving specific values to the parameters

4. Example: stack.adb, stack.ads, stack_test.adb

   Observations

   [stack.ads]

   a. This specifies a generic package, which must be instantiated specifically. The generic formal parameters are specified between the words "generic" and "package".

      i. The formal parameters can be either types or values. (This example has one of each).

      ii. It is possible to specify a default value for a formal parameter. (This is true of subprogram parameters as well.) The default is used if the instantiation does not specify a different value.

   b. When the package is instantiated, and use of Item_Type or Maximum_Size in either the specification or the body will be replaced by the corresponding actual parameter (or default value.)

   c. It is possible to overload a name to refer to either a procedure or a function with the same number of parameters (Pop). The compiler determines which to use by context (is the value being used.)

   [stack.adb]

   d. It is possible to declare a variable as belonging to a subrange (Top_Ind).

   [stack_test.adb]

   DEMO

   e. Instantiation of a generic package. In this case, one of the parameters is specified, and a default is used for the other.

   f. In Ada, parameters can be specified by position or by name.

   h. Note that the name of the instantiated package is used as a variable. That is, the type My_Stack is the whole package, and hence it has a Top_Ind and Item. The operations of the package are invoked by My_Stack. (Compare notion of creating an instance of a class.)

IV. Exception Handling in Ada

A. Another key issue modern programming languages must face is providing for the orderly handling of errors and other unusual events that occur during program execution.

   1. PL/1 was the first major language to incorporate such facilities; but the approach it took has been replaced by a simpler, cleaner approach in newer languages.

   2. CLU incorporated an approach to exception handling that has since been picked up (with variations) by Ada, C++, and Java. We will look at the Ada approach here.

B. One of the major goals of Ada was to develop a language suitable for programming embedded real-time systems. Such an environment demands that a program must never crash - no matter what may happen. Further, the program must be able to recover gracefully from all manner of corrupted data which may be generated by malfunctioning sensors or environmental noise. The exception-handling features of Ada attempt to address this problem.

   In Ada, exceptions may be signalled in one of three ways:

   1. The Ada implementation itself will raise certain kinds of pre-defined exceptions - namely:

      Constraint_Error: an attempt is made to violate a constraint on an object - e.g.

```
I: Integer range 0 .. 100;
...

I := 200;
```

      (Note: a smart compiler might catch this particular one, but such violations are most often only catchable at run time.)

      In Ada 95, division by zero, or an overflow or underflow occurred in an arithmetic operation is also treated as a Constraint_Error. (Ada 83 had another exception type called Numeric_Error, which was removed from Ada 95)

      (Note: an implementation is not required to always catch overflow; but if it does, this is the exception that will be raised. Violations of range constraints and division by zero are always caught.)

      Select_Error: all alternative entries of a select statement in a task are closed (their guards are false) - this means that deadlock has occurred. (We'll discuss this when we get to tasking)

      Storage_Error: an attempt to allocate dynamic storage overflows the available space.

      Tasking_Error: can occur during inter-task communication

2. The standard IO library (e.g. Text_IO) will raise certain exceptions - e.g. Data_Error if the user types an "A" when he is supposed to type a number.

3. Any user code may raise an exception by executing a raise statement - e.g.

   raise Data_Error

C. As we have noted, the Ada language itself defines certain exception names, and the IO library defines others. The programmer may himself define additional exceptions in the declarations part of a unit

exception_demo.adb

DEMO - show valid, negative number, overflow (20), invalid integer, Control-D exit

Observations

1. Defined exceptions

2. Read_It catches and handles formatting problems - converts Data_Error or Constraint_Error to Bad_Integer. Use of | in when clause of handler allows one handler to deal with both types of exception.

3. Factorial checks for a negative integer first. To catch overflow, it needs to explicitly check since the hardware does not catch overflow.

4. Since most exceptions should not terminate the program, an inner block is created in the main program whose exception handler catches bad integers, negative integers, and overflow.

5. The handler for the main block catches end of file.

6. Use of a when others clause.

V. Support for OO in Ada 95

A. Recall, from CS112, that we said that a truly object-oriented programming language must support all aspects of the OO "PIE"

1. ASK

   Polymorphism, Inheritance, Encapsulation.

2. A language that supports polymorphism and encapsulation but not inheritance is said to the OBJECT-BASED. Ada83 is such a language.

B. One of the additions made to Ada 95 was support for inheritance, which makes Ada fully capable of supporting OO software development (though it is still largely used for software developed using the procedural paradigm.)

   employees.ads, .adb; oo_demo.adb

   Observations

1. A class and its subclasses are typically in the same package. Record structures are defined for the base class and the subclasses.

   a. A base class is declared as a "type <base class> is abstract tagged record ...".

   b. A subclass is declared as "type <subclass> is new <base class> with record ...".

   c. A subclass record has all the fields of the parent record, plus the new ones.

2. One defines defines subprograms that operate on the class and its subclasses. The parameter type allows overriding - e.g. there is a Put for Employees that is overridden by versions for each of the two subclasses.

   a. A base-class subprogram can be declared abstract if each subclass implements it.

      (Example: Weekly_Pay).

   b. A base-class subprogram can be invoked by the sub-class subprograms by using a type conversion syntax (cf super. in Java)

      (Examples: Get and Put subclass implementations in employees.adb)

   c. Of course, any base-class subprogram that is not overridden is inherited (and may be used) by the subclass subprograms.

      (Example: Put_Name)

3. It is possible to declare a polymorphic pointer type that can refer to any member of a type hierarchy, by using the syntax "access all <base class>'Class".

   Example: declaration of Employee_Ptr in employees.ads

   a. A new object can be created, and a variable of such a type can be made to point it, by using new

      Example: first line of cases in OO_Demo procedure implementation

   b. A variable of such a type is dereferenced by using the syntax "variable.all"

      Example: second line of cases in OO_Demo procedure implementation

   c. In the case of overridden methods, the correct version to call is determined dynamically based on the _actual_ type of the pointer, rather than its declared type.

      Example: Even though New_Ptr is declared as a pointer to the base class Employee, The calls to Get invoke the methods declared in the subclasses Hourly_Employee or Salaried_Employee based on the actual type of the object the pointer references. Note that dynamic binding is done based on the dynamic type of a _parameter_.

      Ada can do dynamic dispatching based on the type of any parameter, but does not do dynamic dispatching based on the types of two different parameters with differing types. Thus, its capabilities are really no more powerful than those of more "standard" OO languages that dynamically dispatch on the type of the "this" parameter.