Programming Languages

Lecture 14: Data Abstraction

Benjamin J. Keller

Department of Computer Science, Virginia Tech

Overview

- Motivation for data abstractions
- Abstract Data Types
- Modules and early objects
- Object-Oriented Languages

Problems With Writing Large Programs

- Wulf and Shaw: Global Variables Considered Harmful (1973)
  1. Side effects — accesses hidden in functions
  2. Indiscriminant access — can't control access
  3. Screening — may lose access via new declaration of variable
  4. Aliasing
- Characteristics of solution:
  1. No implicit inheritance of variables
  2. Right to access by mutual consent
  3. Access to structure does not imply access to substructure
  4. Provide different types of access (e.g., read-only)
  5. Decouple declaration, name access, and allocation of space

Abstract Data Types

- A major thrust of programming language design in 1970's
- Package data structure and its operations in same module
- Data type consists of set of objects plus set of operations on the objects of the type (constructors, accessors, destructors)
- Want mechanism to build new data types (extensible types)
- Should be treated same way as built-in types
- Representation should be hidden from users (abstraction)
- Users only have access via operations provided by the ADT (encapsulation)
- Distinguish between specification and implementation

ADT Specification

- ADT specification declares data type and operations without implementation details, but possibly with semantics of operations
- Provides information needed to use ADT in programs
- Typically includes
  1. Data structures: constants, types, and variables accessible to user (although details may be hidden)
  2. Declarations of functions and procedures accessible to user (bodies not provided)

ADT Specification (cont)

- May also specify behavioral obligation of an implementation
- As an example, an algebraic specification of behavior of a stack might look like

```
pop(push(S,x)) = S,
if not empty(S) then push(pop(S), top(S)) = S
```

- Formal specification of ADTs uses *universal* algebras

  Data + Operations + Equations = Algebra

ADT Implementation (Representation)

- Definition of the implementation details of an ADT collected in one location
- Usually not accessible to user — some form of access control
- Provides details on all data structures (including some hidden to users) and bodies of all operations

ADTs and Programming Methodology

- Top-down design — partition solution of problem into tasks that can be solved by procedures or functions.
- ADT methodology is orthogonal to top-down design
  - First, identify data types and build modules corresponding to ADTs
  - Use top-down within ADTs and in program using ADTs

Representation in Language

- Three predominant concerns in language design:
  - Simplicity of design
  - Application of formal techniques to specification and verification
  - Keep down lifetime costs
- Reusable modules to represent ADTs quite important.
  - Separate (but not independent) compilation.
  - Want to maintain type checking
  - Control over export and import of names (scope)

Simula 67

- Simulation language derived from Algol 60
- First language to include notion of class
  - Each kind of object being simulated belongs to a class
  - Objects called class instances
  - Class similar to type but includes procedures, functions, and variables
- Representation not hidden — attributes are publicly available

Simula 67 Example

```
class vehicle(weight,maxload);
    real weight, maxload;
begin
    integer licenseno;  (* attributes of class instance *)
    real load;
    Boolean procedure tooheavy;
        tooheavy := weight + load > maxload;
    load := 0;      (* initialization code *)
end
```

Refer to objects through references:

```
    ref(vehicle) rv, pickup;
    rv:- new vehicle(2000,2500);
    pickup:- rv;       (* special assignment via sharing *)
    pickup.licenseno := 3747;
    pickup.load := pickup.load +150;
    if pickup.tooheavy then ...
```

A Detour

- Problems occur when defining a new type in terms of existing types
- Example: represent rational numbers as records (or ordered pairs) of integers
  1. Representation might have several values that do not correspond to any values of the desired type: `(3,0)`
  2. Representation might have multiple values corresponding to the same abstract value: `(1,2)`, `(2,4)`, etc.
  3. Values of the new type can be confused with values of the representation type

Abstract Data Types (again)

- Abstract data type is one that is defined by group of operations (including constants) and (possibly) a set of equations
- Set of values only defined indirectly: can be generated by operations, starting from constructors or constants
- Example: Stack defined by EmptyStack, push, pop, top, and empty operations and equations

$$\begin{aligned}
\text{pop}(\text{push}(fst, rest)) &= rest, \\
\text{top}(\text{push}(fst, rest)) &= fst, \\
\text{empty}(\text{EmptyStack}) &= true, \\
\text{empty}(\text{push}(fst, rest)) &= false, \\
& etc.
\end{aligned}$$

- Key is representation is hidden.

Ada (approx 1979)

- Designed via a U.S. DoD competition
- Packages used to define abstract data types
- Package together type, operations (and state) and hide representation
- Provides support for parameterized packages (polymorphism)

Ada Packages

```
package <package-name> is
   -- declarations of visible types, variables,
   -- constants, and subprograms
 private
   -- complete definitions of private types and
   -- constants
end <package-name>;

package body <package-name> is
    -- definitions of local variables, types,
    -- and subprograms, and complete bodies for
    -- subprograms declared in the specification
    -- part above.  Code for initialization and
    -- exception handlers
end <package-name>;
```

Ada Sample Program

```
package VECT_PACKAGE is  -- declarations only
    type REAL_VECT is array (INTEGER range <>) of float;
    function SUM(V: in REAL_VECT) return FLOAT;
    procedure VECT_PRODUCT(V1,V2 : in REAL_VECT) return
    function MAX(V: in REAL_VECT) return FLOAT;
end VECT_PACKAGE ;

package body VECT_PACKAGE is  -- details of implementation
    function SUM(V: in REAL_VECT) return FLOAT is
        TEMP : FLOAT := 0.0;
    begin
        for I in V'FIRST..V'LAST loop
            TEMP:= TEMP + V(I);
        end loop;
        return TEMP;
    end;
    -- definitions of VECT_PRODUCT and MAX subprograms would go here
end VECT_PACKAGE ;
```

Ada Sample Program (cont)

```
with VECT_PACKAGE, TEXT_IO;                    -- used to make separately compiled package visible
procedure MAIN is
    use VECT_PACKAGE, TEXT_IO;                 -- eliminates need for qualifiers
    package INT_IO is new INTEGER_IO(INTEGER); --instantiation of generic packages
    package REAL_IO is new FLOAT_IO(FLOAT);
    use INT_IO, REAL_IO;
    K: INTEGER range 0..99;
begin
    loop
        GET(K);
        exit when K<1;
        declare                                -- start of block
            A : REAL_VECT(1..K);               -- provides subscript bounds
        begin
            for J in 1..K loop
                GET(A(J));
                PUT(A(J));
            end loop;
            PUT("SUM = ");
            PUT(SUM(A));                       -- uses package function
        end;                                   -- end of block
    end loop;
end MAIN ;
```

Generic Stack Example

Stack represented internally in package (closer to object-oriented than get with Modula 2)

```
generic
    length : Natural := 100;      -- generic parameters
    type element is private;
    -- only assignment and tests for = may be done on
    -- objects of "private" type
    -- "limited private" is also available.
package stack is
    procedure push (X : in element);
    procedure pop (X: out element);
    function empty return boolean;
    stack_error : exception;
end stack;
```

Generic Stack Example (Body)

Data structure of stack is entirely hidden from user — there is no object of type stack available to user.

```
package body stack is
    space   : array (1..length) of element;
    top : integer range 0..length := 0;

    procedure push (X : in element) is
    begin
        if full()  then
            raise stack_error;
        else
            top := top + 1;
            space(top) := X;
        end if;
    end push;
```

Generic Stack Example (Body — cont)

```
    procedure pop (X: out element) is
    begin
        if empty() then
            raise stack_error;
        else
            X := space(top);
            top := top - 1;
        end if;
    end pop;

    function empty return boolean is
    begin
        return (top = 0);
    end;

    function full return boolean is
    begin
        return (top = length);
    end;

end stack;
```

Using Stack Example

```
package stack1 is new stack(20,integer);
package stack2 is new stack(100, character);
    -- Note that length in both cases initialized to 0
use stack2;
stack1.push(5)
if not stack1.empty() then
    stack1.pop(Z);
endif;
push('z');
```

Ada Packages

- Package definition is very much like that of a record with procedures allowed as (non-updatable) fields.

```
stack =  package
           push : procedure (X : in element);
           pop : procedure (X: out element);
           empty : function return boolean;
           stack_error : exception;
         end package;
```

- One of the key ideas behind object-oriented programming

More Ada Packages

- Can also have package where user can manipulate objects of type stack (external representation) like in Modula-2.
- Advantage to external representation is that can define array of stack, etc.
- Biggest problem is exposure of representation of ADT in private part of package specification
- Not available to user, but must recompile if change representation

Generic Stack Again

```
generic
        length : Natural := 100;
        type element is private; -- generic parameters
package stack is
        type stack is private;
        procedure make_empty (S : out stack);
        procedure push (S : in out stack; X : in element);
        procedure pop (S : in out stack; X: out element);
        function empty (S : stack)  return boolean;
        stack_error : exception;
private
        type stack is record
                space   : array(1..length) of element;
                top     : integer range 0..length := 0;
        end record;
end stack;
```

More Ada Packages

```
package body stack is

        procedure make_empty (S : out stack);
        begin
                S.top := 0;
        end make_empty ;

        procedure push (S : in out stack; X : element) is
        begin
                if full(S)  then
                        raise stack_error;
                else
                        S.top := S.top + 1;
                        S.space(S.top) := X;
                end if;
        end push;
```

More Ada Packages

```
        procedure pop (S : in out stack; X: out element) is
        begin
                if empty(S) then
                        raise stack_error;
                else
                        X := S.space(S.top);
                        S.top := S.top - 1;
                end if;
        end pop;

        function empty(S : in out stack) return boolean is
        begin
                return (top = 0);
        end;

        function full(S : in out stack) return boolean is
        begin
                return (S.top = length);
        end;

end stack;
```

Modula-2

- Modula (modular language) designed by Wirth in 1975 for programming small real-time control systems (no files, sets, or pointers)
- Modula-2 is 1980 revision (influenced by Mesa at Xerox PARC) intended to synthesize systems programming with general purpose language supporting modern software engineering
- Operating system for Lilith microcomputer written in Modula 2
- Minor changes to Pascal to simplify programming (reliability) and improve program readability and efficiency
- Upward extension of Pascal to support large system design projects (Separately compiled and type checked modules)
- Downward extension to allow machine-level programming and supports coroutines

Sample Modula-2 Program

```
DEFINITION MODULE stackMod;
IMPORT element FROM elementMod;
        TYPE stack;
        PROCEDURE make_empty (VAR S : stack);
        PROCEDURE push (VAR S : stack; X : element);
        PROCEDURE pop (VAR S : stack; X: element);
        PROCEDURE empty (S : stack): BOOLEAN;
END stackMod.

IMPLEMENTATION MODULE stackMod;
        TYPE stack = POINTER TO RECORD
                space   : array[1..length] of element;
                top     : INTEGER;
        END;

        PROCEDURE make_empty (VAR S : stack);
        BEGIN
                S^.top := 0;
        END make_empty ;

        ...
END stackMod;
```

Modula-2 (cont)

- *Opaque* types (those declared but undefined in Definition module) must be pointers or take no more space than pointers
- Compare and contrast Modula-2 and Ada on supporting abstract data types
  - Both can import from other units (modules or packages) and export items to other units
  - For external representations not much difference
  - Private types in Ada force recompilation if implementation changes, but opaque types in Modula-2 do not
  - Internal representations almost identical
  - Generics make Ada more flexible — can use to create new instances of packages

Clu (1974)

- *Cluster* is basis of support for abstract data types
- Provides both packaging and hiding of representation (cvt used to go back and forth between external abstract type and internal representation).
- May have parameterized clusters where specify needed properties of type parameter.

```
sorted_bag = cluster [t : type] is create, insert, ...
             where t has
                   lt, equal : proctype (t,t) returns (bool);
```

- Biggest difference from Ada and Modula-2 is that cluster is a type. Therefore can create multiple copies. Elements of cluster types are held as implicit references.

ML Abstype

- ADT supported in very straightforward way in ML
- Provides for encapsulation and information hiding
- Example:

```
abstype intstack = mkstack of (int list)
  with exception stackUnderflow
    val emptyStk = mkstack []
    fun push (e:int) (mkstack(s):intstack) = mkstack(e::s)
    fun pop (mkstack([])) = raise stackUnderflow
      | pop (mkstack(e::s)) = mkstack(s)
    fun top (mkstack([])) = raise stackUnderflow
      | top (mkstack(e::s)) = e
    fun IsEmpty (mkstack([])) = true
      | IsEmpty (mkstack(e::s)) = false
    end;
```

Generic stack ADT in ML

```
abstype 'a stack = mkstack of ('a list)
  with exception stackUnderflow
    val emptyStk : 'a stack = mkstack []
    fun push (e:'a) (mkstack(s):'a stack) = mkstack(e::s)
    fun pop (mkstack([]):'a stack) = raise stackUnderflow
      | pop (mkstack(e::s):'a stack) = mkstack(s):'a stack
    fun top (mkstack([]):'a stack) = raise stackUnderflow
      | top (mkstack(e::s):'a stack) = e
    fun IsEmpty (mkstack([]):'a stack) = true
      | IsEmpty (mkstack(e::s):'a stack) = false
   end;
```

- Cannot get at representation of stack
- Reference to `mkstack(l)` will generate an error message
- Only access through constants and operations
- Modules more sophisticated support with separate compilation

Object-Oriented Programming Languages

- Roots in languages supporting ADT's
- Biggest loss in moving from FORTRAN to Pascal is lack of support for modules with persistent local data.
- Clu, Ada, and Modula 2 attempted to remedy this by adding clusters, packages, and modules.
- In Ada and Modula 2, objects (i.e. packages, and modules) were late additions to an earlier paradigm (Pascal-like)
  - couldn't be instantiated dynamically
  - had no type or other method for organizing, despite similarity to records.
  - nonetheless provide better modules for building large systems
- Called object-based languages.

Goals of OO Languages

- Want to support development of high-quality software
- Qualities Desired in Software
  - *Correctness*: guarantee correctness of code on legal input
  - *Robustness*: work in unusual cases, handle errors
  - *Extensibility*: ability to add features
  - *Reusability*: ability to use code in different circumstances
  - *Compatibility*: ability to combine modules
- ADT languages provide reasonable support for all but extensibility, some limitations on reusability.
- Object-oriented languages are an attempt to make progress toward these goals.

Definition of OOL

A programming language is object-oriented if:

1. It supports objects that are data abstractions
   (Like Modula 2, Ada)
2. All objects have an associated object type (called classes)
   (Different from Modula 2, Ada)
3. Classes may inherit attributes from superclasses
   (Very different from Modula 2, Ada)
4. Computations proceed by sending messages to objects
5. Routines may be applied to objects which are variants of those they are designed to be applied to (subtype polymorphism)
6. Support dynamic method invocation (explained later)

Object Oriented Languages

- Simula 67 first object-oriented language — designed for discrete event simulation
- Until recently, Smalltalk best-known — by Alan Kay at Xerox
  Several versions: Smalltalk-72,-74,-76,-78,-80.
- C++, object-oriented extensions to Pascal, C, LISP, etc.
- Eiffel designed as OOL — Bertrand Meyer
  (book *Object-Oriented Software Construction*)

Object-Oriented Programming

- Main idea of object-oriented programming:

  Independent objects cooperate to perform a computation by sending messages to each other

- Object-oriented languages built around following concepts:
  - Object — like internal representation of abstract data types all data encapsulated in objects — first class data
  - Message — request for object to carry out one of its operations
  - Class — template for set of objects (similar to type)
  - Instance — object described by a class
  - Method — operation associated with an object — specified in class

Object-Oriented Programming

- Subtype — A is a subtype of B (A $\Leftarrow$ B) if A represents a specialization of B

  Example: `cars` $\Leftarrow$ `vehicles`, `ColorPoint` $\Leftarrow$ `Point`

  An element of a subtype can be used in any context in which an element of the supertype would work

- Subclass — An incremental modification or extension of a class and its methods. Methods not changed are inherited.

Objects

- Objects are internal data abstractions — only accessible to outer world through associated procedures
  - hide representation from outer world
  - have an associated state (current contents of internal variables)
  - its methods have automatic access to its state, therefore no explicit parameter needed for self.

Object Types & Classes

- Object types
  - allow objects (modules) to be first-class
  - allow use in assignment, parameters, components of structures
  - allow objects to be classified via subtyping
- Classes
  - templates for the creation of objects
  - contain definitions of all methods
  - can create subclasses that inherit methods

Current Language Characteristics

- Most current OOLs identify object types and classes — especially subtypes and subclasses
- This identification can lead to holes in type system and/or restrictions in expressibility
- In typical object-oriented programming language, everything is an object
- Preserves abstraction — no object can make changes to the internal state of another object, must send messages using methods in public interface.

  Not enforced by some languages

Type Problems in Eiffel

- Allowable changes which can be made in subclasses:
  1. Can add new features: instance variables or routines
  2. Instance variables may be given a new type which is a subclass of the original
  3. When redefining routines, possible to replace parameter and result types by subclasses of originals

     Automatic in inherited routines if type defined as `like Current`
- More flexible than Object Pascal or C++
- Big problem with Eiffel is identification of class with type

Problem with Class/Type Identification

- C' is a *subclass* of C if C' inherits from C
- Thus C' inherits attributes and methods from superclass
- May redefine methods in subclass so that replace class of arguments and return value by subclasses
- Example: if `m(a:`A`):`B in C then can redefine `m(a:`A'`):`B' in subclass C' if A' inherits from A and B' inherits from B
- Can lead to holes in typing system: subtype $\neq$ subclass
- A' is a *subtype* of A if element of type A' can be used where element of type A is expected
- Can use subclass element anywhere superclass element is expected.

Type Problem Example

```
class LINKABLE [G]

feature

        item: G;

        right: like Current;  -- Right neighbor

        put_right (other: like Current) is
        -- Put ‘other’ to the right of current cell.
                do
                        right := other
                ensure
                        chained: right = other
                end;

end -- class LINKABLE
```

Type Problem Example (cont)

Now define subclass:

```
class BI_LINKABLE [G] inherit

        LINKABLE [G]
                redefine
                        put_right
                end

feature -- Access
        left: like Current;
                        -- Left neighbor

        put_right (other: like Current) is
                        -- Put ‘other’ to the right of current cell.
                do
                        right := other;
                        if (other /= Void) then
                                other.simple_put_left (Current)
                        end
                end;
```

Type Problem Example (cont)

```
put_left (other: like Current) is
                -- Put ‘other’ to the left of current cell.
        do
                left := other;
                if (other /= Void) then
                        other.simple_put_right (Current)
                end
        ensure
                chained: left = other
        end;

simple_put_right (other: like Current) is
                -- set ‘right’ to ‘other’
        do
                if right /= Void then
                        right.simple_forget_left;
                end;
                right := other
        end;
```

Type Problem Example (cont)

```
        simple_put_left (other: like Current) is
                        -- set ‘left’ to ‘other’ is
                do
                        if left /= Void then
                                left.simple_forget_right
                        end;
                        left := other
                end;

invariant

        right_symmetry:
                (right /= Void) implies (right.left = Current);
        left_symmetry:
                (left /= Void) implies (left.right = Current)

end -- class BI_LINKABLE
```

Type Problem Example (cont)

- But now suppose have following routine

```
trouble(p, q : LINKABLE [RATIONAL] ) is
    do
        p.put_right(q);
        ....
    end
```

- and also

```
s_node : LINKABLE [RATIONAL]
bi_node: BI_LINKABLE [RATIONAL]
```

- What happens when write `trouble(bi_node,s_node)`
- If `BI_LINKABLE [RATIONAL]` is subtype of `LINKABLE [RATIONAL]`, then this should work...
- Instead results in system crash

Problem

- Routine `s_node.put_right` takes a parameter of type (class)

  `LINKABLE [RATIONAL]`

- Routine `bi_node.put_right` takes a parameter of type (class):

  `BI_LINKABLE [RATIONAL]`

- These are not subtypes because

$$A' \rightarrow B' \Leftarrow A \rightarrow B \text{ iff } B' \Leftarrow B \text{ and } A \Leftarrow A'$$

Another Problem

- Export method from superclass, but not from subclass.
- This will break system if send message to object of subclass which is not visible
- Example:

```
hide_n_break(a:A) is
    do
        a.meth ....
    end
```

and write `hide_n_break(b)` where `b :` A', and A' is subclass of A that does not export `meth`

Problems Solved?

- Earlier versions of Eiffel allowed user to break the type system in these ways
- Eiffel 3.0 mandates global check of all classes used in a system to make sure that above situation could not occur
- One consequence is that a system could work fine, but addition of new (separately compiled) class could break a previously defined class
- Of course, compilers have to implement this validity check
- In Fall, '95, Bertrand Meyer announced solutions to "covariant typing problem" at OOPSLA '95. Soon after hole in solution was found.
- Other problems may exist

Type Problems in OOLs

Most statically typed object-oriented languages are one of following

1. Type unsafe like Eiffel
2. Too rigid with types (like C++ or Object Pascal) forcing programmers to bypass the type system.

   Example: C++ doesn't allow user to change type of method arguments (can change return type), but also has many type system holes (e.g., unchecked casts)
3. Insert dynamic checks where unsafe (Java, Beta).

Avoiding Problem

- Only allow subclasses that are subtypes (Trellis/Owl)
- Restrictive because rules out classes that are reasonable extensions of superclass
- Example: `ColorPoint` class derived from `Point` adding color

Better Solution

- Distinguish subtype and inheritance hierarchies
- Inheritance hierarchy has only to do with implementation
- Subtype hierarchy has only to do with interface
- Therefore class does not equal type
- Allows multiple classes generating objects of same type: Cartesian and polar points with same external interface

Details

- Still need restrictions on redefinitions to avoid breaking other inherited methods:

```
method1(...) = ... p.method2(..)....
method2(...) = .....
```

- If redefine `method2` with different type, how do we know it will continue to be type-safe when used in `method1` (presuming `method1` is inherited and not changed)?
- Can set up type-checking rules for legal subclasses and subtypes that give guarantee that type system can't be broken

Pros of Object-Oriented Languages

1. Good use of information hiding. Objects can hide their state.
2. Good support for reusability. Especially with support for generics like Ada, with run-time creation of objects (unlike Ada).
3. Support for inheritance and subtyping provides for reusability of code.

Cons of OOLs: Inheritance

- Loss of locality with inheritance.
- Definition of class dependent on others.
- Changes in superclasses may have big impact on descendents.
- Semantics of inheritance is very complex.
  - Small changes in methods may make major changes in semantics of subclass.
  - Must know definition of methods in superclass to predict impact of changes in subclass.
  - Makes providing libraries more complex.

Cons of OOLs: Type-Checking

- Type-checking too rigid, unsafe, or requires link time global analysis.
- Interface can break when linked with new customer class.
- Can be avoided by restricting inheritance or separating subtyping and inheritance
- Some languages insert run-time checks to cover lack of type-safety.

Impact of Object-Oriented Languages

- History of adoption without careful assessment of consequences. Now growing reaction against C++.
- Many programmers don't really understand paradigm
- Most of the advantages possible in older languages: Clu, Modula-2, or Ada
- Fewer typing problems with languages that provide subtyping without inheritance: Modula-3, Haskell, Quest, etc.
- Advice: carefully specify meaning of methods, avoid long inheritance chains, avoid inheritance only for reuse sake, and be careful of interactions of methods.
- Java could be a very successful compromise between flexibility and usefulness if add parameterized classes.