

Program Examples - 5th Edition

These examples were developed for the 5th edition of Software Engineering and many of them have been used as the basis of the Java examples in the 6th edition. However, there is not usually a simple correspondence between the examples so, rather than try and link examples with specific chapters in the 6th edition, I have commented each of the examples here with information about where it is relevant in the 6th edition.

I do not have (and I have no plans to develop) Ada or C++ implementations of any Java examples that do not have equivalent Ada/C++ examples from the 5th edition.

Sixth Edition - Chapter 9 - Formal Specification

These are specifications of a List abstract data type. The formal algebraic specification of a list is given on p199.

```
generic
-- A private generic type means assignment and equality must be
-- defined on that type
    type Elem is private ;
package List is
    type T is private ;
-- Create operation is implicit. Lists created by declaration
    procedure Head (L: T ; V: out Elem ; Err: out ERROR_INDICATOR) ;
-- Length can't fail so no need for error indicator
    function Length (L: T) return NATURAL ;
    procedure Tail (L: T ; LT: out T ; Err: out ERROR_INDICATOR) ;
-- Cons can't fail so no need for error indicator
    function Cons (L: T ; V: Elem ) return T ;
private
-- an Ada access type corresponds to a Pascal pointer
-- the entity referenced by the pointer is defined in the package body
-- In this case, it would be a record with one field pointing to the next
-- list element
    type LISTREC ;
    type T is access LISTREC ;
end List ;
```

Figure unused - Ada list specification

```
// assume error-indic has been declared as an enumerated
// type and the name List has been declared

template <class elem> class ListRec {
    friend class List <elem> ;
private:
    Elem val ;
    ListRec <elem>* next ;
    // Does this have to be a pointer rather than an element itself - see below
    ListRec ( const elem val, const ListRec <elem>* next) ;
}

template <class elem> class List {
public:
    // List initialises the list to nil pointer.
    List ()
    // Does this have to return a pointer to an elem rather than an elem because
```

```
// the compiler doesn't know what store to allocate?  
// &Err means call by reference. Check this out.  
elem Head (error-indic &Err) ;  
int Length ( ) ;  
// This should be OK as the storage for list is known  
// Is the <elem> classifier needed here or just List?  
List<elem> Tail (error-indic &Err) ;  
List <elem> Cons (const elem V) ;  
private:  
    ListRec <elem>* First ;  
}
```

Figure unused - C++ list specification

Sixth Edition - Chapter 12 - Object-oriented Design

The example of the weather station that runs through this chapter was completely redesigned from the 5th to the 6th edition. Therefore, there is no real correspondence between the program fragments used as illustrations.

```
typedef struct instrument_status {
    int in1, in2 ;
};

typedef struct weather_data_rec {
    int wd1, wd2 ;
};

typedef enum {comp1, comp2} computer_id ;

class Weather_station {
public:
    Weather_station () ;
    ~Weather_station () ;
    void Transmit_data (computer_id dest) ;
    void Transmit_status (computer_id dest) ;
    void Self_test () ;
    void Shut_down () ;
    // Access and constructor functions
    char* Identifier () ;
    void Put_identifier (char* Id) ;
    instrument_status Inst_status () ;
    void Put_instrument_status (Instrument_status ISD) ;
    weather_data_rec Weather_data () ;
    void Put_weather_data (weather_data_rec WDR) ;
private:
    char* station_identifier ;
    weather_data_rec Weather_data ;
    instrument_status inst_status ;
};
```

Figure unused - C++ specification of the weather station Class

Sixth Edition - Chapter 14 - Design for reuse

The examples here were not used in the 6th edition as I changed the focus of the text from design for reuse to design with reuse. These examples are intended to illustrate general purpose components that have been explicitly designed for reuse.

```
generic
  type ELEMENT is private ;
package Linked is
  -- Exported type declarations
  type LIST is limited private ;
  type STATUS is range 1..10 ;
  type ITERATOR is private ;

  -- Comparison operations
  function Equals (L1, L2: LIST) return BOOLEAN ;
  function Equivalent (L1, L2: LIST) return BOOLEAN ;

  -- access operation
  -- true if the list has no elements
  function Is_empty (L: LIST) return BOOLEAN ;
  -- returns the number of elements in the list
  function Size_of (L: LIST) return NATURAL ;
  -- true if a list element is the same as E
  function Contains (E: ELEMENT; L: LIST)
    return BOOLEAN ;
  -- returns the first list element
  procedure Head (L: LIST; E: in out ELEMENT ;
    Error_level: out STATUS) ;
  -- removes the first list element and returns the remaining list
  procedure Tail (L: LIST; Outlist: in out LIST ;
    Error_level: out STATUS) ;

  -- Constructor operations (Fig. 20.7)
  -- Append adds an element to the end of the list
  procedure Append ( E: ELEMENT; Outlist: in out LIST ;
    Error_level: out STATUS) ;
  -- Add adds an element to the front of the list
  procedure Add ( E: ELEMENT; Outlist: in out LIST ;
    Error_level: out STATUS) ;
  -- Add_before adds an element before element value E
  procedure Add_before ( E: ELEMENT ; Outlist: in out LIST ;
    Error_level: out STATUS) ;
  -- Add_after adds an element after element E
  procedure Add_after ( E: ELEMENT; Outlist: in out LIST ;
    Error_level: out STATUS) ;
  -- Replace replaces the element matching E1 with E2
  procedure Replace ( E1, E2: ELEMENT; Outlist: in out LIST ;
    Error_level: out STATUS) ;
  -- Clear deletes all members of a list
  procedure Clear ( Outlist: in out LIST ;
    Error_level: out STATUS) ;
  -- Prune removes the last element from the list
  procedure Prune ( Outlist: in out LIST ;
    Error_level: out STATUS) ;
  -- Prune_to deletes the list up to and including
  -- the element matching E
  procedure Prune_to ( E: ELEMENT; Outlist: in out LIST ;
    Error_level: out STATUS) ;
  -- Prune_from deletes list after element matching E
  procedure Prune_from( E: ELEMENT; Outlist: in out LIST ;
```

```

    Error_level: out STATUS ) ;
-- Remove deletes the element which matches E
procedure Remove ( E: ELEMENT; Outlist: in out LIST ;
    Error_level: out STATUS ) ;
-- Remove_before and Remove_after delete the element before
-- and after E respectively
procedure Remove_before ( E: ELEMENT; Outlist: in out LIST;
    Error_level: out STATUS ) ;
procedure Remove_after ( E: ELEMENT; Outlist: in out LIST ;
    Error_level: out STATUS ) ;

-- I/O operations
procedure Print_list (L: LIST; Error_level: out STATUS ) ;
procedure Write_list (F: TEXT_IO.FILE_TYPE ; L: LIST;
    Error_level: out STATUS ) ;
procedure Read_list (F: TEXT_IO.FILE_TYPE ;
    Outlist: out LIST ; Error_level: out STATUS ) ;

-- Iterator operations (Fig. 20.9)
procedure Iterator_initialise (L: LIST; Iter: in out ITERATOR;
    Error_status: in out STATUS) ;
procedure Go_next (L: LIST; Iter: in out ITERATOR;
    Error_status: in out STATUS) ;
procedure Eval (L: List; Iter: in out ITERATOR;
    Val: out ELEMENT; Error_status: in out STATUS) ;
function At_end (L: LIST; Iter: ITERATOR) return BOOLEAN ;
private
    type LIST_ELEM;
    type LIST is access LIST_ELEM ;
    type ITERATOR is access LIST_ELEM ;
end Linked ;

```

Figure unused - Ada implementation of reusable list abstract data structure

// Go to the text of the chapter to get the details here

Ada

```

package Counter is
    type T is limited private;
    procedure Inc (Cnt : in out T);
    procedure Dec (Cnt : in out T);
    procedure Copy (Cnt1: T ; Cnt2: out T);
    function Cequals (Cnt1, Cnt2: T) return BOOLEAN ;
private
    type T is range 0..500_000 ;
end Counter;

```

C++

```

class Counter {
public:
    Counter Inc () ;
    Counter Dec () ;
    // needs to be implemented as friends. Look into this- how do these work
    void Copy (Counter c2) ;
    // should this be an operator ??
    boolean Eq (Counter c2) ;
private:
    int value ;
} // Counter

```

Figure unused - A portable counter component

```
Ada
package Process is
  type PID is private ;
  type STATUS is (READY, RUNNING, WAITING, KILLED)
  function Create return PID ;
  function Kill (Process: PID) return STATUS ;
  function Get_status (P: PID) return STATUS ;
  function Wake_up (P: PID) return STATUS ;
  function Put_to_sleep (P: PID) return STATUS ;
  -- wait causes process to wait until given status ?? Check text here
  procedure Wait (P: PID; S: STATUS) ;
private
  type PID is new NATURAL ;
end Process_management ;

C++
enum P-state { READY, RUNNING, WAITING, KILLED } ;
class Process {
public:
  Process () ;
  p-state Kill () ;
  p-state Get () ;
  p-state Wake-up () ;
  p-state Put-to-sleep () ;
  // check what this does
  void Wait (p-state status) ;
private:
  p-state status ;
}
```

Figure unused - A process management package

Sixth Edition - Chapter 18 - Critical systems development

The following examples of queue specifications in Ada and C++ were used as examples of information hiding. A Java interface example that is closely related is given in Figure 18.2.

```
package Queue is
  type T is private ;
  procedure Put (IQ : in out T; X: INTEGER);
  procedure Remove (IQ : in out T; X: out INTEGER);
  function Is_empty (IQ : T ) return BOOLEAN;
private
  type Q_RANGE is range 0..100 ;
  type Q_VEC is array (1..100) of INTEGER ;
  type T is record
    The_queue: Q_VEC ;
    front, back : Q_RANGE ;
  end record;
end Queue;
```

Figure unused - An integer queue declaration as an Ada package specification

```
class IntQueue {
public:
  IntQueue () ;
  Put ( int x ) ; // Adds an item to the queue
  int Remove () ; // this has side effect of changing queue
  boolean Is_empty () ;
private:
  int front, back ;
  int qvec [100] ;
}
```

Figure unused - An integer queue declaration as a C++ class declaration

```
generic
  type ELEM is private ;
  type Q_SIZE is range <> ;
package Queue is
  type T is private ;
  procedure Put (IQ : in out T; X: ELEM);
  procedure Remove (IQ : in out T; X: out ELEM);
  function Is_empty (IQ : in T ) return BOOLEAN;
private
  type Q_VEC is array (Q_SIZE) of ELEM ;
  type T is record
    The_queue: Q_VEC ;
    Front : Q_SIZE := Q_SIZE'FIRST ;
    Back: Q_SIZE := Q_SIZE'FIRST ;
  end record;
end Queue;
```

Figure unused - An Ada generic specification of a queue

```

template
  <class elem>
class Queue {
public:
  Queue (int qsize) ;
  // Should this take a pointer to elem
  Put ( elem x ) ; // Changes the queue
  // Does this have to return a pointer to elem?
  elem Remove ( ) ; // this has side effect of changing queue
  boolean Is_empty ( ) ;
private:
  int front, back ;
  elem qvec [ ] ; // Is this syntax OK. Size set in initialisation function
}

```

Figure unused - C++ generic specification of a queue

```

type IQ_SIZE is range 0..49 ; type LQ_SIZE is range 0..199 ;
package Integer_queue is new Queue (ELEM => INTEGER,
                                     Q_SIZE => IQ_SIZE ) ;
package List_queue is new Queue (ELEM => List.T,
                                   Q_SIZE => LQ_SIZE ) ;

Queue <int> IntQueue (50) ;
Queue <List> ListQueue (200) ;
// Remember to discuss the differences between them in the text.
// Explain differences between packages and classes

```

Figure unused - Generic instantiation in Ada and C++

The following examples correspond with Figure 18.6 - the freezer controller.

```

with Pump, Temperature_dial, Sensor, Globals, Alarm;
use Globals ;

procedure Control_freezer is
  Ambient_temperature: FREEZER_TEMP ;
  -- Check this syntax!
  Danger_temp: fixed constant := -18.0 ;
begin
  loop
    Ambient_temperature := Sensor.Get_Temperature ;
    if Ambient_temperature > Temperature_dial.Setting then
      if Pump.Status = Off then
        Pump.Switch (State => On) ;
      end if ;
      if Ambient_temperature > -18.0 then
        raise Freezer_too_hot ;
      end if ;
      elsif Pump.Status = On then
        -- Switch pump off because temperature is low
        Pump.Switch (State => Off) ;
      end if ;
    end loop ;
exception
  when Freezer_too_hot => Alarm.Activate ;
  raise ;
  when others => Alarm_activate ;
  raise Control_problem ;
end Control_freezer;

```

Figure 18.6a An Ada implementation of a temperature controller with exceptions

```

// Stroustrup, Chapter 9 for exception handling
// C++ has parameterised exceptions. Better than Ada

class Freezer-too-hot {}; //Exception classes.
class Control-problem {};

// not sure if interface should list exceptions or not. Seems good practice
void Control_freezer () throw (Freezer-too-hot, Control-problem)
{
    float Ambient_temp ;
    const float Danger_temp = -18.0 ;
    // try means exceptions will be handled in this block
    try {
        while (true) {
            Ambient_temp = Sensor.Get-temperature () ;
            if (Ambient_temp > Temperature_dial.Setting () )
            {
                if (Pump.Status () = off)
                    Pump.Switch (on) ;
                if ( Ambient_temp > Danger_temp )
                    throw Freezer-too-hot ( ) ;
            }
            else
                if (Pump.Status () = on)
                    Pump.Switch (off) ;
        } //while loop
    } // end of exception handling try block
    // catch indicates the exception handling code.
    catch (Freezer-too-hot) { // Is this syntax right?
        Alarm.Activate () ;
        throw ; // no need to repeat exception name
    }
    // Ellipses (...) means catch all other exceptions
    Catch (...) {
        Alarm.Activate () ;
        throw Control-problem () ;
    }
} // Control_freezer

```

Figure 18.6b A C++ implementation of a temperature controller with exceptions

Positive_even below corresponds with Figure 18.7.

```

package Positive_even is
  type NUMB is limited private ;
  procedure Assign (A: in out NUMB; B: NATURAL;
                   State_error: in out BOOLEAN) ;
  function Eval (A: NUMB) return NATURAL ;
  function Eq (A, B: NUMB) return BOOLEAN ;
private
  type NUMB is new NATURAL ;
end Positive_even ;

class Positive-even {
public:
    friend Positive-even Assign ( int b, state-error &err ) ;
    int Eval () ;
    // This should be declared as a friend of Positive-even -somehow - look at friends
    // boolean Eq (Positive-even b) ;

```

```
private:
    int numb ;
} //Positive-even
```

Figure 18.7a Even number abstract data type in Ada and C++

```
procedure Assign (A: in out NUMB; B: NATURAL;
                  State_error: in out BOOLEAN) is
begin
    if B rem 2 /= 0 then
        State_error := TRUE ;
    else
        State_error := FALSE ;
        A := NUMB ( B ) ;
    end if ;
end Assign ;

Positive-even:: Assign (int b, state-error &err)
{
    if (b%2 != 0)
        err = true ;
    else {
        err = false ;
        numb = b ;
    }
} //Positive-even::Assign
```

Figure 18.7b Assertion checking in Ada and C++

These examples are related to Figure 18.8 - a n array implementation that can be checked for damage.

```
package Checked_array is
    type ELEM is range 1..64 ;
    type INDEX is NATURAL range 1..6 ;
    type T is private ;
    procedure Check (A: in out T) ;
    function Is_damaged (A: T) return BOOLEAN ;
    function Eval (A: T; I: INDEX) return ELEM ;
    -- Assign must check that input is a power of 2 < 128
    function Assign (A: T; I: INDEX; E: ELEM) return T;
private
    type SHORT_ARRAY is array (INDEX) of ELEM ;
    type T is record
        A: SHORT_ARRAY ;
        Damaged: BOOLEAN ;
    end record ;
end Checked_array ;
```

Figure 18.8a Damage assessment in an Ada abstract data type specification

```

template <class elem> class short-array {
private:
    short-array (int I) ;
    elem vals [] ; //check this syntax for array declarations whose size is allocated
}

template
    <class elem>
class Checked-array {
public:
    checked-array () ;
    void Check () ;
    boolean Is-damaged () ;
    // Should this be a pointer - see earlier problem
    elem Eval ( int I ) ;
    // Must check that E is a power of 2 < 128
    void Assign ( int I ; elem E ) ;
private:
    boolean damaged ;
    short-array <elem> Values ( 6 ) ;
}

```

Figure 18.8b Damage assessment in C++

The next example corresponds to Figure 18.9, a sort program with built-in checks that the sort has worked correctly.

```

procedure Sort (X: in out ELEM_ARRAY ) is
    Copy: ELEM_ARRAY ;
begin
    -- Take a copy of the array to be sorted.
    for i in ELEM_ARRAY'RANGE loop
        Copy (i) := X (i) ;
    end loop ;
    -- Code here to sort the array X in ascending order
    -- Now test that the array is actually sorted
    for i in ELEM_ARRAY'FIRST..ELEM_ARRAY'LAST-1 loop
        if X (i) > X (i + 1) then
            -- a problem has been detected - raise exception
            raise Sort_error ;
        end if ;
    end loop ;
exception
    -- restore state and indicate to calling procedure
    -- that a problem has arisen
    when Sort_error =>
        for i in ELEM_ARRAY'RANGE loop
            X (i) := Copy (i) ;
        end loop ;
        raise ;
    -- unexpected exception. Restore state and indicate
    -- that the sort has failed
    when Others =>
        for i in ELEM_ARRAY'RANGE loop
            X (i) := Copy (i) ;
        end loop ;
        -- should raise a different kind of exception here
        raise Sort_error;
end Sort ;

```

Figure 18.9a Ada sort procedure with backward error recovery

```

Sort (elem-array X ) {
    elem-array copy ;
    int i = 0 ;
    // Take a copy of the array to be sorted.
    for ( all elements in the array ) {
        Copy (i) = X (i) ;
        i++ ; // should this be in copy expressions
    }
    try {
        // Code here to sort the array X in ascending order - not shown
        // Now test that the array is actually sorted
        for (all elements in X, referenced by i)
            if (X (i) > X (i + 1) )
                -- a problem has been detected - raise exception
                throw Sort_error ;
    } // try block
    -- restore state and indicate to calling procedure
    -- that a problem has arisen
    catch (Sort_error ) {
        for(all elements in X, indexed by i) {
            X (i) = Copy (i) ;
            i++ ;
        }
        throw ;
    }
    -- unexpected exception. Restore state and indicate
    -- that the sort has failed
    catch (...) {
        for(all elements in X, indexed by i) {
            X (i) = Copy (i) ;
            i++ ;
        }
        throw Unexpected_error;
    } // Sort ;
}

```

Figure 18.9b C++ sort procedure with backward error recovery

// Still unfinished but some code deliberately missing as I may change to a more
// abstract description. The descriptions are really too similar.

Sixth Edition - Chapter 20 - Software Testing

The examples here correspond to Figure 20.9, a Java implementation of a binary search function.

```
procedure Binary_search (Key: ELEM ; T: ELEM_ARRAY ;
  Found: in out BOOLEAN ; L: in out ELEM_INDEX ) is
  -- Preconditions
  -- T'FIRST <= T'LAST and
  -- forall i: T'FIRST..T'LAST-1, T (i) <= T(i+1)

  Bott : ELEM_INDEX := T'FIRST ;
  Top : ELEM_INDEX := T'LAST ;
  Mid : ELEM_INDEX;
begin
  L := (T'FIRST + T'LAST) mod 2;
  Found := T ( L ) = Key;
  while Bott <= Top and not Found loop
    Mid := (Top + Bott) mod 2;
    if T( Mid ) = Key then
      Found := true;
      L := Mid;
    elsif T( Mid ) < Key then
      Bott := Mid + 1;
    else
      Top := Mid - 1;
    end if;
  end loop;
end Binary_search;
```

Figure 23.10 Implementation of binary search routine

Do I need this in C++ as well

```
void binary_search (elem key, elem-array T, bool &found, int &L)
{
  int bott = T.first (), top = T.last (), mid ;
  // Assume first and last are functions in class elem-array
  // they have their obvious meanings
  L := ( top+bott) / 2 ;
  found = if (T(L)=key)? true: false ;
  while (Bott <=Top and not found)
  {
    Mid = Top+Bott / 2 ;
    if (T(Mid)=Key) {
      found = true;
      L = Mid ;
    }
    else if (T(Mid) < Key)
      Bott = Mid + 1 ;
    else
      Top = Mid-1 ;
  } // while
} //binary_search
```