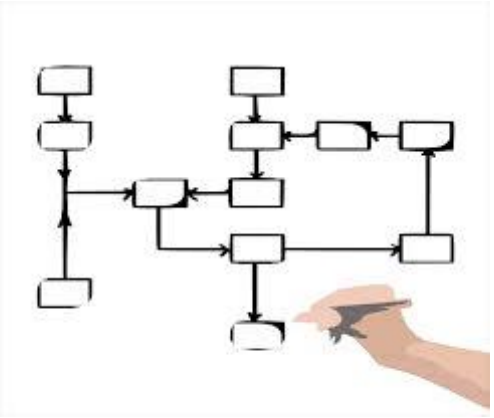
A Brief History of Modularity in Programming



“Programming in the Large” Steps

Design & Implement

- Program & programming style (done)
- Common data structures and algorithms (done)
- Modularity **<--** we are here
- Building techniques & tools (done)

Debug

- Debugging techniques & tools (done)

Test

- Testing techniques (done)

Maintain

- Performance improvement techniques & tools

Goals of this Lecture

Help you learn:

- The history of modularity in computer programming
- A rational reconstruction of the development of programming styles, with a focus on modularity

Why? Modularity is important

- Abstraction is a powerful (the only?) technique available for understanding large, complex systems
- A power programmer knows how to find the abstractions in a large program
- A power programmer knows how to convey a large program's abstractions via its modularity

And also… History is important

- Only by understanding the past can we fully appreciate the present

Agenda

Non-modular programming

Structured programming (SP)

Abstract object (AO) programming

Abstract data type (ADT) programming

Non-Modular Programming

Example languages

- Machine languages
- Assembly languages
- FORTRAN (**For**mula **Tran**slating System)
- BASIC (**B**eginners **A**ll-Purpose **S**ymbolic **I**nstruction **C**ode)

Non-Modular Example

Example program

- Dr. Dondero's first computer program
- 1971
- Teletype terminal
- Survived on paper

Functionality

- Help fellow algebra students learn how to expand **poly**nomials
- Name: POLLY.BAS

Design

- BASIC language
- Don't be concerned with details…

Non-Modular Example

POLLY.BAS

```
5   PRINT "IF YOU NEED INSTRUCTIONS TYPE 0."; (1)
7   INPUT X (2)
8   IF X=0 THEN 10 (3)
9   IF X#0 THEN 60
10   PRINT "HELLO! THIS PROGRAM IS DESIGNED TO GIVE YOU PRACTICE" (4)
11   PRINT "IN EXPANDING, THROUGH THE USE OF THE DISTRIBUTIVE" (5)
12   PRINT "PROPERTY.  IT WILL ALSO HELP YOU TO OVERCOME THE" (6)
13   PRINT "FRESHMAN MISTAKE.  PLEASE RESPOND TO EACH QUESTION" (7)
14   PRINT "BY TYPING THE NUMBER OF THE ANSWER CORESPONDING TO" (8)
15   PRINT "THAT QUESTION." (9)
27   PRINT (10)
28   PRINT (11)
29   PRINT (12)
30   PRINT TAB(21)"LIST OF ANSWERS" (13)
40   PRINT "************************************************************" (14)
50   PRINT TAB(1)"1. -4A^2 - 2A^2 + 2A^2B"; (15)
51   PRINT TAB(36)"4. -4A^2 + 2A^2 + 2A^2B" (16)
52   PRINT TAB(1)"2. -4A^2 -2A^2 -2A^2B"; (17)
53   PRINT TAB(36)"5. 4A^2 - 2A^2 -2A^2B" (18)
54   PRINT TAB(1)"3. -A^2 - A - AB"; (19)
55   PRINT TAB(36)"6. -2A^2 + 2a + 2AB" (20)
```

Non-Modular Example

POLLY.BAS (cont.)

```
56   PRINT (21)
57   PRINT (22)
58   PRINT (23)
60   PRINT "OK! HERE WE GO!!!" (24)
61   PRINT (25)
62   PRINT (26)
63   PRINT (27)
70   PRINT "EXPAND:"; (28)
71   GOSUB 8000 (29)
72   GOTO 90 (32)
73   GOSUB 8010 (54 end trace)
74   GOTO 141
75   GOSUB 8020
76   GOTO 170
77   GOSUB 8030
78   GOTO 200
79   GOSUB 8040
80   GOTO 300
81   GOSUB 8050
82   GOTO 400
```

```
90   PRINT "WHAT IS YOUR ANSWER? "; (33)
100  INPUT A (34) (43)
110  IF A=1 THEN 550 (35) (44)
115  IF A=2 THEN 550 (45)
120  IF A=3 THEN 780 (46)
125  IF A=4 THEN 550
130  IF A=5 THEN 550
135  IF A=6 THEN 550
140  IF A#6 THEN 9990
141  PRINT "WHAT IS YOUR ANSWER? ";
150  INPUT B
155  IF B=1 THEN 580
156  IF B=2 THEN 580
158  IF B=3 THEN 580
160  IF B=4 THEN 580
162  IF B=5 THEN 580
164  IF B=6 THEN 800
166  IF B#6 THEN 9990
```

Non-Modular Example

POLLY.BAS (cont.)

```
170   PRINT "WHAT WILL IT BE THIS TIME? ";
175   INPUT C
178   IF C=1 THEN 620
180   IF C=2 THEN 820
182   IF C=3 THEN 620
184   IF C=4 THEN 620
186   IF C=5 THEN 620
188   IF C=6 THEN 620
190   IF C#6 THEN 9990
200   PRINT "WHAT IS YOUR GUESS? ";
210   INPUT D
214   IF D=1 THEN 660
216   IF D=2 THEN 660
218   IF D=3 THEN 660
220   IF D=4 THEN 840
222   IF D=5 THEN 660
224   IF D=6 THEN 660
226   IF D#6 THEN 9990
```

```
300   PRINT "WHAT IS YOUR ANSWER? ";
310   INPUT E
314   IF E=1 THEN 860
316   IF E=2 THEN 700
318   IF E=3 THEN 700
320   IF E=4 THEN 700
322   IF E=5 THEN 700
324   IF E=6 THEN 700
326   IF E#6 THEN 9990
400   PRINT "WHAT WILL IT BE? ";
410   INPUT F
414   IF F=1 THEN 740
416   IF F=2 THEN 740
418   IF F=3 THEN 740
420   IF F=4 THEN 740
422   IF F=5 THEN 880
424   IF F=6 THEN 740
426   IF F#6 THEN 9990
```

Non-Modular Example

POLLY.BAS (cont.)

```
550  GOSUB 9000 (36)
570  GOTO 100 (42)
580  GOSUB 9000
600  GOTO 150
620  GOSUB 9000
640  GOTO 175
660  GOSUB 9000
680  GOTO 210
700  GOSUB 9000
720  GOTO 310
740  GOSUB 9000
760  GOTO 410
780  GOSUB 9010 (47)
785  GOSUB 9020 (50)
790  GOTO 73 (53)
800  GOSUB 9010
805  GOSUB 9020
810  GOTO 75
820  GOSUB 9010
825  GOSUB 9020
830  GOTO 77
840  GOSUB 9010
845  GOSUB 9020
850  GOTO 79
860  GOSUB 9010
865  GOSUB 9020
870  GOTO 81
880  GOSUB 9010
890  GOTO 9998
```

```
8000  PRINT "-A(A + 1 + B)" (30)
8001  RETURN (31)
8010  PRINT "-2A(A - 1 - B)"
8011  RETURN
8020  PRINT "-2A(2A + A + AB)"
8021  RETURN
8030  PRINT "-2A(2A - A - AB)"
8031  RETURN
8040  PRINT "-(4A^2 + 2A^2 -2A^2B)"
8041  RETURN
8050  PRINT "-A(-4A + 2A + 2AB)"
8051  RETURN
9000  PRINT "YOUR ANSWER IS INCORRECT." (37)
9005  PRINT "LOOK CAREFULLY AT THE SAME PROBLEM AND GIVE" (38)
9006  PRINT "ANOTHER ANSWER." (39)
9007  PRINT "WHAT WILL IT BE? "; (40)
9008  RETURN (41)
9010  PRINT "YOUR ANSWER IS CORRECT." (48)
9015  RETURN (49)
9020  PRINT "NOW TRY THIS ONE.” (51)
9030  RETURN (52)
9990  PRINT "THAT'S NOT A REASONABLE ANSWER."
9991  PRINT "COME BACK WHEN YOU GET SERIOUS."
9992  GOTO 9999
9998  PRINT "SORRY, THIS IS THE END OF THE PROGRAM."
9999  END
```

Toward SP

What's wrong?

- From programmer's viewpoint?

Think about

- Flow of control

Toward SP (Bőhm & Jacopini)

Bőhm and Jacopini

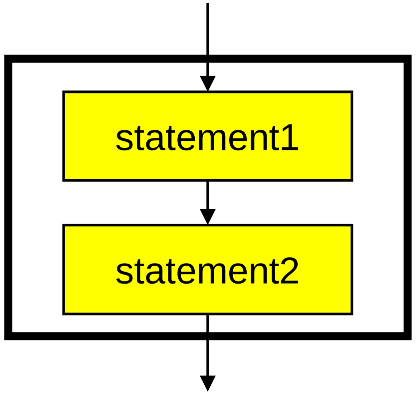
Any algorithm can be expressed as the nesting of only 3 control structures: sequence, selection, repetition

Corrado Bőhm

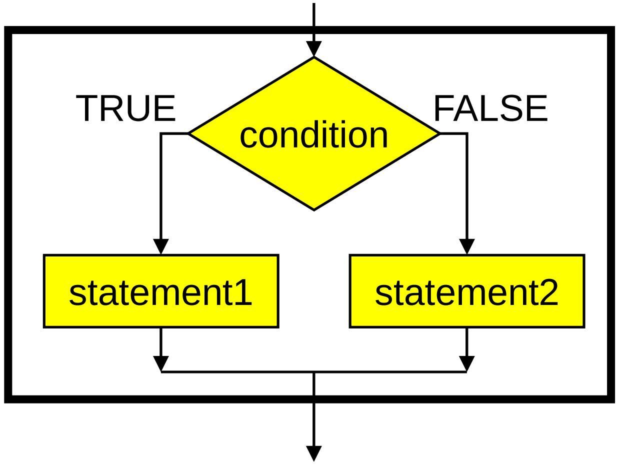
Corrado Bőhm and Guiseppe Jacopini.
"Flow diagrams, Turing machines and languages with only two formation rules."
Communications of the ACM 9 (May 1966), 366-371.

Toward SP (Bőhm & Jacopini)

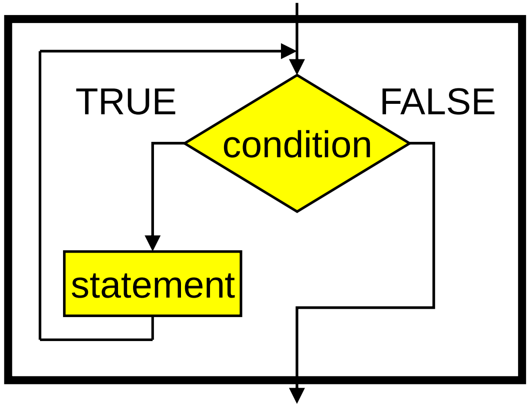
Sequence



Selection



Repetition



Toward SP (Dijkstra)

Edsger
Dijkstra

Edsger Dijkstra.
"Go To Statement Considered Harmful."
Communications of the ACM, Vol. 11,
No. 3, March 1968, pp. 147-148.

Toward SP (Dijkstra)

Paraphrasing (quotes in Appendix 1) …

A program

- Is a **static** entity
- Has no time dimension

A process

- Is a program in execution
- Is a **dynamic** entity
- Has a time dimension
- Can be understood only in terms of its time dimension

People understand **static** things better than they understand **dynamic** things

So the **static** structure of a program should be similar to its **dynamic** structure

Toward SP (Dijkstra)

Or, in other words…

Suppose:

- We have program written on paper 1
- Each time computer executes a statement, we write that statement on paper 2

Then consider the correspondence between paper 1 and paper 2

- Conditionals interfere, but only slightly
- Function calls interfere
- Iterations interfere

Nevertheless, for the sake of clarity...

Toward SP (Dijkstra)

Paper 2 should be similar to paper 1

- The **dynamic** rep of the program should be similar to the **static** rep of the program

And ...

- If the static rep of the program contains goto statements, then paper 2 will be dissimilar to paper 1

So avoid goto statements

Toward SP

Bőhm & Jacopini:

- Any program **can** be expressed as the nesting of only 3 control structures

Bőhm & Jacopini + Dijkstra

- Any program **should** be expressed as the nesting of only 3 control structures

Agenda

Non-modular programming

Structured programming (SP)

Abstract object (AO) programming

Abstract data type (ADT) programming

Structured Programming

Key ideas:

- Programming using only the nesting of the 3 elementary control structures: sequence, selection, iteration
- (Arguably) occasional exceptions are OK
- Define functions/procedures/subroutines liberally

Example languages:

- Pascal
- C

Example program…

- (Don't be concerned with details)

SP Example

rev.c

Functionality

- Read numbers (doubles) from **`stdin`** until end-of-file
- Write to **`stdout`** in reverse order

Design

- Use a stack (LIFO data structure) of doubles
- Represent stack as an array
- To keep things simple…
  - Assume max stack size is 100
  - (See precept examples for more realistic implementations)

SP Example

rev.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

enum {MAX_STACK_ITEMS = 100}; /* Arbitrary */

int push(double *stack, int *top, double d)
{   assert(stack != NULL);
    assert(top != NULL);
    if (*top >= MAX_STACK_ITEMS)
       return 0;
    stack[*top] = d;
    (*top)++;
    return 1;
}

double pop(double *stack, int *top)
{   assert(stack != NULL);
    assert(top != NULL);
    assert(*top > 0);
    (*top)--;
    return stack[*top];
}
```

SP Example

rev.c (cont.)

```
int main(void)
{   double stack[MAX_STACK_ITEMS];
    int top = 0;
    double d;
    while (scanf("%lf", &d) == 1)
       if (! push(stack, &top, d))
          return EXIT_FAILURE;
    while (top > 0)
       printf("%g\n", pop(stack, &top));
    return 0;
}
```

Toward AO Programming

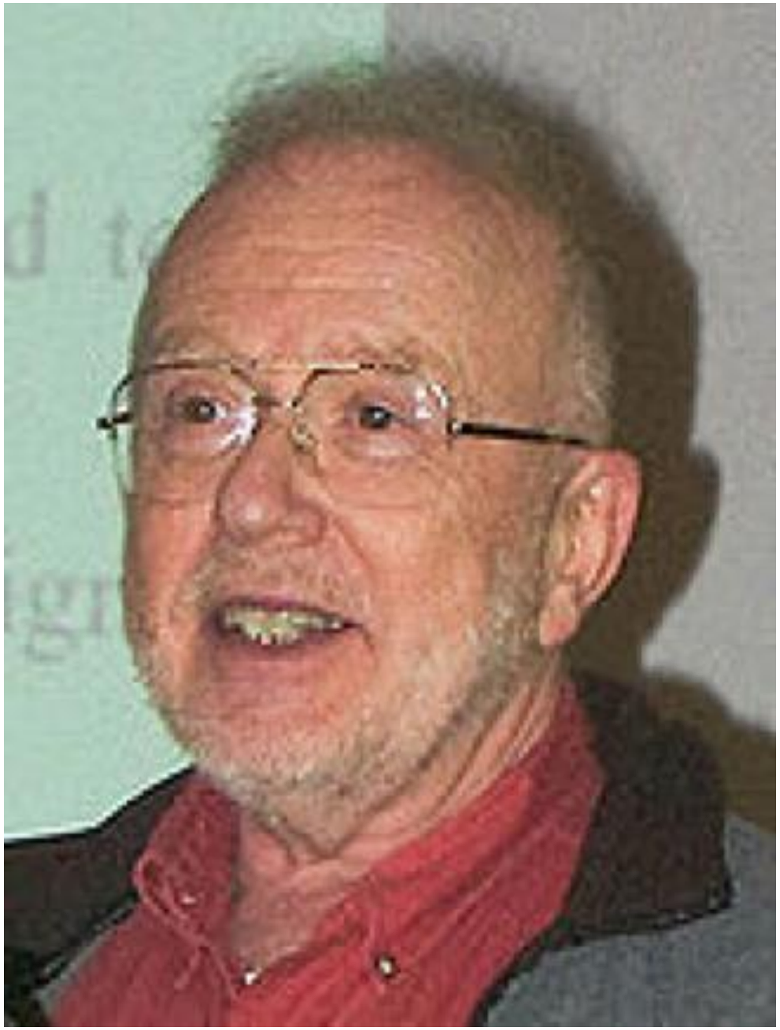
What's wrong?

- From programmer's viewpoint?

Think about:

- Design decisions
- Modularity

Toward AO Programming



David
Parnas

David Parnas
"On the Criteria to be Used in Decomposing Systems into Modules."
Communications of the ACM, Vol. 15, No. 12, December 1972. pp. 1053 – 1058.

Agenda

Non-modular programming

Structured programming

Abstract object (AO) programming

Abstract data type (ADT) programming

Abstract Object Programming

Key ideas:

- Design modules to encapsulate important design decisions
- Design modules to hide info from clients

Example languages

- Ada
- C (with some discipline)

Example program…

AO Programming Example

stack.h (interface)

```
#ifndef STACK_INCLUDED
#define STACK_INCLUDED

int    Stack_init(void);
void   Stack_free(void);
int    Stack_push(double d);
double Stack_pop(void);
int    Stack_isEmpty(void);

#endif
```

AO Programming Example

rev.c (client)

```
#include "stack.h"
#include <stdio.h>
#include <stdlib.h>

int main(void)
{  double d;
   Stack_init();
   while (scanf("%lf", &d) == 1)
      Stack_push(d);
   while (! Stack_isEmpty())
      printf("%g\n", Stack_pop());
   Stack_free();
   return 0;
}
```

For simplicity, error handling code is omitted

AO Programming Example

stack.c (implementation)

```
#include "stack.h"
#include <assert.h>

enum {MAX_STACK_ITEMS = 100};

static double stack[MAX_STACK_ITEMS];
static int top;
static int initialized = 0;

int Stack_init(void)
{   assert(! initialized);
    top = 0;
    initialized = 1;
    return 1;
}

void Stack_free(void)
{   assert(initialized);
    initialized = 0;
}
```

```
int Stack_push(double d)
{   assert(initialized);
    if (top >= MAX_STACK_ITEMS)
        return 0;
    stack[top] = d;
    top++;
    return 1;
}

double Stack_pop(void)
{   assert(initialized);
    assert(top > 0);
    top--;
    return stack[top];
}

int Stack_isEmpty(void)
{   assert(initialized);
    return top == 0;
}
```

AO Programming Example

Notes:

- One Stack **object**
- The Stack object is **abstract**
  - Major design decision (implementation of Stack as array) is hidden from client
  - Client doesn't know Stack implementation
  - Change Stack implementation => need not change client
- Object state is implemented using global variables
  - Global variables are **`static`** => clients cannot access them directly

Toward ADT Programming

What's wrong?

- From programmer's viewpoint?

Think about

- Flexibility

Toward ADT Programming

Barbara
Liskov

Toward ADT Programming

"An **abstract data type** defines a class of abstract objects which is completely characterized by the operations available on those objects. This means that an abstract data type can be defined by defining the characterizing operations for that type."

Barbara Liskov and S. Zilles. "Programming with Abstract Data Types." ACM SIGPLAN Conference on Very High Level Languages. April 1974.

Agenda

Non-modular programming

Structured programming

Abstract object (AO) programming

Abstract data type (ADT) programming

ADT Programming

Key ideas:

- A module should be **abstract**
  - As in AO programming
- A module can (and often should) be a **data type**!!!
  - Data type consists of data and operators applied to those data
  - Program can create as many objects of that type as necessary

Example languages

- CLU (ALGOL, with **clu**sters)
- C++, Objective-C, C#, Java, Python
- C (with some discipline)

Example program…

ADT Programming Example

stack.h (interface)

```
#ifndef STACK_INCLUDED
#define STACK_INCLUDED

enum {MAX_STACK_ITEMS = 100};

struct Stack
{   double items[MAX_STACK_ITEMS];
    int top;
};

struct Stack *Stack_new(void);
void          Stack_free(struct Stack *stack);
int           Stack_push(struct Stack *stack, double d);
double        Stack_pop(struct Stack *stack);
int           Stack_isEmpty(struct Stack *stack);

#endif
```

ADT Programming Example

rev.c (client)

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

int main(void)
{  double d;
   struct Stack *stack1;
   stack1 = Stack_new();
   while (scanf("%lf", &d) == 1)
      Stack_push(stack1, d);
   while (! Stack_isEmpty(stack1))
      printf("%g\n", Stack_pop(stack1));
   Stack_free(stack1);
   return 0;
}
```

For simplicity, error handling code is omitted

ADT Programming Example

stack.c (implementation)

```
#include <stdlib.h>
#include <assert.h>
#include "stack.h"

struct Stack *Stack_new(void)
{   struct Stack *stack;
    stack = (struct Stack*)malloc(sizeof(struct Stack));
    if (stack == NULL)
       return NULL;
    stack->top = 0;
    return stack;
}

void Stack_free(struct Stack *stack)
{   assert(stack != NULL);
    free(stack);
}
```

ADT Programming Example

stack.c (cont.)

```
int Stack_push(struct Stack *stack, double d)
{   assert(stack != NULL);
    if (stack->top >= MAX_STACK_ITEMS)
        return 0;
    stack->items[stack->top] = d;
    (stack->top)++;
    return 1;
}

double Stack_pop(struct Stack *stack)
{   assert(stack != NULL);
    assert(stack->top > 0);
    stack->top--;
    return stack->items[stack->top];
}

int Stack_isEmpty(struct Stack *stack)
{   assert(stack != NULL);
    return stack->top == 0;
}
```

ADT Programming

What’s wrong?

- From programmer’s viewpoint?

Think about

- Encapsulation

See next lecture!

Summary

A rational reconstruction of the history of modularity in computer programming

- Non-modular programming
- Structured programming (SP)
- Abstract object (AO) programming
- Abstract data type (ADT) programming

More recently:

- Object-oriented programming
  - Smalltalk, Objective-C, C++, C#, Java
- Logic-based programming
  - Prolog
- Functional programming
  - LISP, OCaml
- …

Appendix 1: Quotations

Toward SP (Dijkstra)

Edsger
Dijkstra

Edsger Dijkstra.
"Go To Statement Considered Harmful."
Communications of the ACM, Vol. 11,
No. 3, March 1968, pp. 147-148.

Toward SP (Dijkstra)

"My first remark is that, although the programmer's activity ends when he has constructed a correct **program,** the **process taking place under control of his program is** the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its **dynamic behavior has to satisfy the desired** specifications. Yet, once the program has been made, the 'making' of the corresponding process is delegated to the machine."

Edsger Dijkstra.
"Go To Statement Considered Harmful."
Communications of the ACM, Vol. 11,
No. 3, March 1968, pp. 147-148.

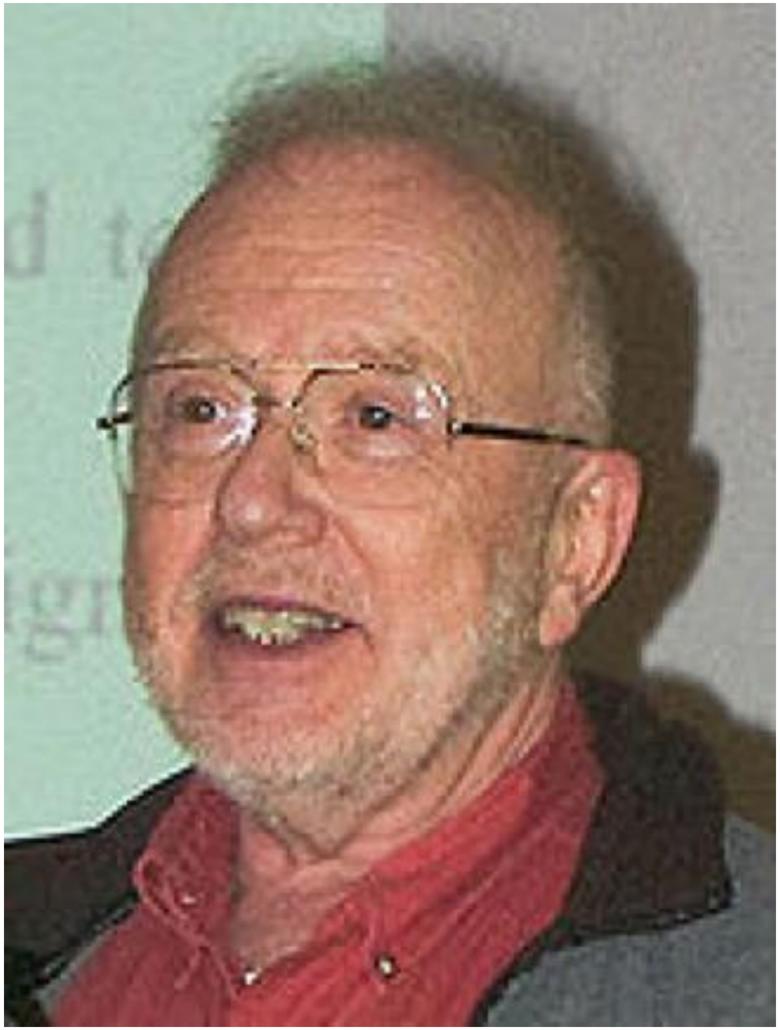
Toward SP (Dijkstra)

"My second remark is that our intellectual powers are rather geared to master **static relations and that our powers to visualize** processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to **make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible."**

Edsger Dijkstra.
"Go To Statement Considered Harmful."
Communications of the ACM, Vol. 11, No. 3, March 1968, pp. 147-148.

Use of the **goto** statement makes the correspondence between the program and the process non-trivial

Toward AO Programming



David Parnas

Toward AO Programming

"In the first decomposition the criterion used was to make **each major step in the processing a module**. One might say that to get the first decomposition one makes a flowchart. This is the most common approach to decomposition or modularization."

David Parnas
"On the Criteria to be Used in Decomposing Systems into Modules."
Communications of the ACM, Vol. 15, No. 12, December 1972. pp. 1053 – 1058.

Toward AO Programming

"The second decomposition was made using '**information hiding' as a criterion.** The modules no longer correspond to steps in the processing... Every module in the second decomposition is characterized by its knowledge of a design decision which it hides from all others. Its interface or definition was chosen to reveal as little as possible about its inner workings."

David Parnas
"On the Criteria to be Used in Decomposing Systems into Modules."
Communications of the ACM, Vol. 15, No. 12, December 1972. pp. 1053 – 1058.

Toward ADT Programming

Barbara Liskov

Toward ADT Programming

"We believe that the above concept captures the fundamental properties of abstract objects. When a programmer makes use of an abstract data object, he is **concerned only with the behavior** which that object exhibits **but not with any details of how that behavior is achieved** by means of an implementation."

Barbara Liskov and S. Zilles.
"Programming with Abstract Data Types."
ACM SIGPLAN Conference on Very
High Level Languages. April 1974.

Toward ADT Programming

"**Abstract types** are intended to be very much like the built-in types provided by a programming language. The user of a built-in type, such as integer or integer array, is only concerned with creating objects of that type and then performing operations on them. He is not (usually) concerned with how the data objects are represented, and he views the operations on the objects as indivisible and atomic when in fact several machine instructions may be required to perform them."

Barbara Liskov and S. Zilles.
"Programming with Abstract Data Types."
ACM SIGPLAN Conference on Very
High Level Languages. April 1974.

Appendix 2: SP Example 2

polly.c

```
#include <stdio.h>
#include <stdlib.h>

static void printInstructions(void)
{  printf("HELLO! THIS PROGRAM IS DESIGNED TO GIVE YOU PRACTICE\n");
   printf("IN EXPANDING, THROUGH THE USE OF THE DISTRIBUTIVE\n");
   printf("PROPERTY.   IT WILL ALSO HELP YOU TO OVERCOME THE\n");
   printf("FRESHMAN MISTAKE.  PLEASE RESPOND TO EACH QUESTION\n");
   printf("BY TYPING THE NUMBER OF THE ANSWER CORESPONDING TO\n");
   printf("THAT QUESTION.\n");
   printf("\n\n\n");
   printf("                     LIST OF ANSWERS\n");
   printf("***************************");
   printf("********************************\n");
   printf("1. -4A^2 - 2A^2 + 2A^2B     4. -4A^2 + 2A^2 + 2A^2B\n");
   printf("2. -4A^2 -2A^2 -2A^2B       5. 4A^2 - 2A^2 -2A^2B\n");
   printf("3. -A^2 - A - AB            6. -2A^2 + 2a + 2AB\n");
   printf("\n\n\n");
}
```

SP Example 2

polly.c (cont.)

```
static void handleSillyAnswer(void)
{  printf("THAT'S NOT A REASONABLE ANSWER.\n");
   printf("COME BACK WHEN YOU GET SERIOUS.\n");
   exit(EXIT_FAILURE);
}

static void handleWrongAnswer(void)
{  printf("YOUR ANSWER IS INCORRECT.\n");
   printf("LOOK CAREFULLY AT THE SAME PROBLEM AND GIVE\n");
   printf("ANOTHER ANSWER.\n");
   printf("WHAT WILL IT BE? ");
}
```

SP Example 2

polly.c (cont.)

```
static int readAnswer(int minAnswer, int maxAnswer)
{   int answer;
    if (scanf("%d", &answer) != 1)
        handleSillyAnswer();
    if ((answer < minAnswer) || (answer > maxAnswer))
        handleSillyAnswer();
    return answer;
}

static void readCorrectAnswer(int correctAnswer)
{   enum {MIN_ANSWER = 1, MAX_ANSWER = 6};
    int answer;
    answer = readAnswer(MIN_ANSWER, MAX_ANSWER);
    while (answer != correctAnswer)
    {   handleWrongAnswer();
        answer = readAnswer(MIN_ANSWER, MAX_ANSWER);
    }
    printf("YOUR ANSWER IS CORRECT.\n");
}
```

SP Example 2

polly.c (cont.)

```
int main(void)
{  int answer;

   printf("IF YOU NEED INSTRUCTIONS TYPE 0. OTHERWISE TYPE 1.\n");
   answer = readAnswer(0, 1);
   if (answer == 0)
      printInstructions();

   printf("OK! HERE WE GO!!!\n");
   printf("\n\n");

   printf("EXPAND:\n");
   printf("-A(A + 1 + B)\n");
   printf("WHAT IS YOUR ANSWER? ");
   readCorrectAnswer(3);

   printf("NOW TRY THIS ONE\n");
   printf("-2A(A - 1 - B)\n");
   printf("WHAT IS YOUR ANSWER? ");
   readCorrectAnswer(6);
```

SP Example 2

polly.c (cont.)

```
    printf("NOW TRY THIS ONE\n");
    printf("-2A(2A + A + AB)\n");
    printf("WHAT WILL IT BE THIS TIME? ");
    readCorrectAnswer(2);

    printf("NOW TRY THIS ONE\n");
    printf("-2A(2A - A - AB)\n");
    printf("WHAT IS YOUR GUESS? ");
    readCorrectAnswer(4);

    printf("NOW TRY THIS ONE\n");
    printf("-(4A^2 + 2A^2 -2A^2B)\n");
    printf("WHAT IS YOUR ANSWER? ");
    readCorrectAnswer(1);

    printf("NOW TRY THIS ONE\n");
    printf("-A(-4A + 2A + 2AB)\n");
    printf("WHAT WILL IT BE? ");
    readCorrectAnswer(5);

    printf("SORRY, THIS IS THE END OF THE PROGRAM.\n");
    return 0;
}
```