“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

Title in retrospect!

Example languages
- Machine languages
- Assembly languages
- FORTRAN (Formula Translating System)
- BASIC (Beginners All-Purpose Symbolic Instruction Code)

Non-Modular Example

Design
- BASIC language
- Don’t be concerned with details…
**Non-Modular Example**

**POLLY.BAS**

```
50 PRINT TAB(36)"5. -2A^2 + 2a + 2AB"
59 PRINT TAB(1)"4. -4A^2 + 2A^2 + 2A^2B"
58 PRINT TAB(1)"3. -A^2 - A - AB"
57 PRINT TAB(36)"2. -4A^2 - 2A^2 - 2A^2B"
56 PRINT TAB(1)"1. -4A^2 - 2A^2 + 2A^2B"
55 PRINT TAB(36)"6. -2A^2 + 2a + 2AB"
```

**Non-Modular Example**

```
28 PRINT
27 PRINT
15 PRINT "THAT QUESTION."
14 PRINT "BY TYPING THE NUMBER OF THE ANSWER CORRESPONDING TO"
13 PRINT "FRESHMAN MISTAKE. PLEASE RESPOND TO EACH QUESTION"
10 PRINT "IN EXPANDING, THROUGH THE USE OF THE DISTRIBUTIVE"
9 IF X#0 THEN 60
7 INPUT X
5 PRINT "IF YOU NEED INSTRUCTIONS TYPE 0.;"
```

**Non-Modular Example**

```
55  PRINT TAB(36)"6. -2A^2 + 2a + 2AB"
54  PRINT TAB(1)"3. -A^2 - A - AB"
53  PRINT TAB(36)"5. 4A^2 - 2A^2 - 2A^2B"
52  PRINT TAB(1)"2. -4A^2 - 2A^2 - 2A^2B"
51  PRINT TAB(36)"4. -4A^2 + 2A^2 + 2A^2B"
50  PRINT TAB(1)"1. -4A^2 - 2A^2 + 2A^2B"
```

**Non-Modular Example**

```
40 PRINT "************************************************************
29 PRINT
28 PRINT
15 PRINT "THAT QUESTION."
14 PRINT "BY TYPING THE NUMBER OF THE ANSWER CORRESPONDING TO"
13 PRINT "FRESHMAN MISTAKE. PLEASE RESPOND TO EACH QUESTION"
10 PRINT "IN EXPANDING, THROUGH THE USE OF THE DISTRIBUTIVE"
9 IF X#0 THEN 60
7 INPUT X
5 PRINT "IF YOU NEED INSTRUCTIONS TYPE 0.;"
```

**Non-Modular Example**

```
55  PRINT TAB(36)"6. -2A^2 + 2a + 2AB"
54  PRINT TAB(1)"3. -A^2 - A - AB"
53  PRINT TAB(36)"5. 4A^2 - 2A^2 - 2A^2B"
52  PRINT TAB(1)"2. -4A^2 - 2A^2 - 2A^2B"
51  PRINT TAB(36)"4. -4A^2 + 2A^2 + 2A^2B"
50  PRINT TAB(1)"1. -4A^2 - 2A^2 + 2A^2B"
```

**Non-Modular Example**

```
40 PRINT "************************************************************
29 PRINT
28 PRINT
15 PRINT "THAT QUESTION."
14 PRINT "BY TYPING THE NUMBER OF THE ANSWER CORRESPONDING TO"
13 PRINT "FRESHMAN MISTAKE. PLEASE RESPOND TO EACH QUESTION"
10 PRINT "IN EXPANDING, THROUGH THE USE OF THE DISTRIBUTIVE"
9 IF X#0 THEN 60
7 INPUT X
5 PRINT "IF YOU NEED INSTRUCTIONS TYPE 0.;"
```

**Non-Modular Example**

```
55  PRINT TAB(36)"6. -2A^2 + 2a + 2AB"
54  PRINT TAB(1)"3. -A^2 - A - AB"
53  PRINT TAB(36)"5. 4A^2 - 2A^2 - 2A^2B"
52  PRINT TAB(1)"2. -4A^2 - 2A^2 - 2A^2B"
51  PRINT TAB(36)"4. -4A^2 + 2A^2 + 2A^2B"
50  PRINT TAB(1)"1. -4A^2 - 2A^2 + 2A^2B"
```

**Toward SP**

What's wrong?
- From programmer's viewpoint?

Think about
- Flow of control

**Toward SP (Böhm & Jacopini)**

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

Corrado Böhm
Toward SP (Böhm & Jacopini)

Sequence

Selection

Repetition

Toward SP (Dijkstra)

Edsger 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.

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.

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

In other words...

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

People understand static things better than they understand dynamic things

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

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 secondarily...
- 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

Böhm & Jacopini:
- Any program can be expressed as the nesting of only 3 control structures
Böhm & 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 literally

Example languages:
- Pascal
- C

Example program...
- (Don’t be concerned with details)

SP Example 1

polly.c

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

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

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

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

SP Example 1 (cont.)

polly.c (cont.)

```c
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?\n");
}
```
SP Example 1

```c
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.
");
}

int main(void)
{
    int answer;
    printf("IF YOU NEED INSTRUCTIONS TYPE 0. OTHERWISE TYPE 1.
");
    answer = readAnswer(0, 1);
    if (answer == 0)
        printInstructions();
    printf("OK! HERE WE GO!!!

");
    printf("EXPAND:
");
    printf("-A(A + 1 + B)
");
    printf("WHAT IS YOUR ANSWER? ");
    readCorrectAnswer(3);
    printf("NOW TRY THIS ONE
");
    printf("-2A(A - 1 - B)
");
    printf("WHAT IS YOUR ANSWER? ");
    readCorrectAnswer(6);
    printf("NOW TRY THIS ONE
");
    printf("-2A(2A + A + AB)
");
    printf("WHAT WILL IT BE THIS TIME? ");
    readCorrectAnswer(2);
    printf("NOW TRY THIS ONE
");
    printf("-2A(2A - A - AB)
");
    printf("WHAT IS YOUR GUESS? ");
    readCorrectAnswer(4);
    printf("NOW TRY THIS ONE
");
    printf("-(4A^2 + 2A^2 -2A^2B)
");
    printf("WHAT IS YOUR ANSWER? ");
    readCorrectAnswer(1);
    printf("NOW TRY THIS ONE
");
    printf("-A(-4A + 2A + 2AB)
");
    printf("WHAT WILL IT BE? ");
    readCorrectAnswer(5);
    printf("SORRY, THIS IS THE END OF THE PROGRAM.
");
    return 0;
}
```

SP Example 2

```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];
}
```

```c
#include "stdlib.h"
#include "stdio.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 2

```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];
}
```
Toward AO Programming

What's wrong?
- From programmer's viewpoint?

Think about:
- Design decisions
- Modularity

David Parnas

"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."

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."

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)

```c
#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
```

rev.c (client)

```c
#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
");
    Stack_free();
    return 0;
}
```

AO Programming Example

stack.c (implementation)

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

#define 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;
}
```

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

Barbara Liskov
“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.”

“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.”

“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.”

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 ADT 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 clusters)
• C++, Objective-C, C#, Java, Python
• C (with some discipline)

Example program...

ADT Programming Example

```c
#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)**

```c
#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.

**stack.c (implementation)**

```c
#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);
}

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;
}
```

**stack (cont.)**

```c
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++, CR Java
- Logic-based programming
  - Prolog
- Functional programming
  - LISP, OCaml