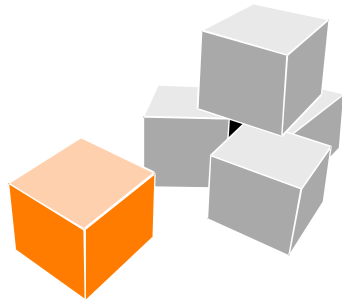


Software Technology Group

TU Darmstadt | FB Informatik

Software Engineering Design

2. A Brief History of Programming

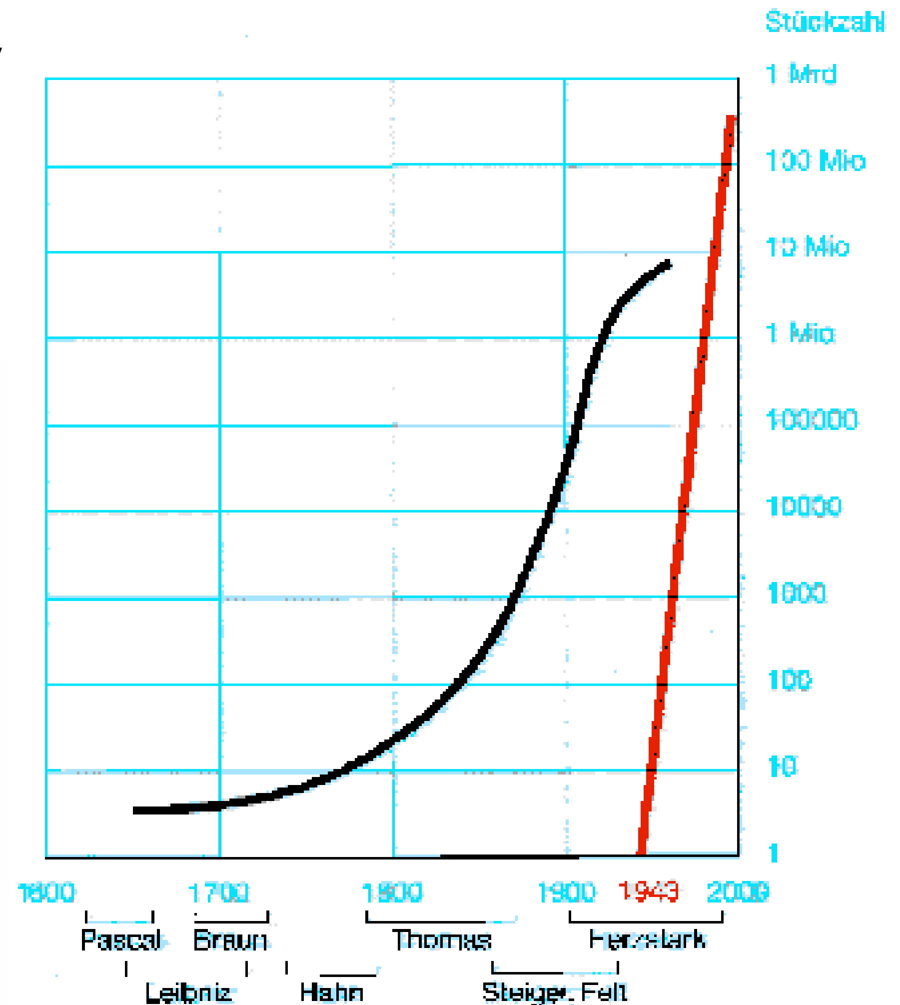
Prof. Dr. Mira Mezini

Dipl.-Inform. Christoph Bockisch

Dipl.-Ing. Michael Haupt

Development of "CS"

- what made this possible?
  - reduction of hardware costs
  - managing the complexity of programming



Entwicklung der Rechenmaschinen im Vergleich
mechanisch
elektronisch

Permanent Software Crisis?

- the software crisis
  - providing understandable, correct, verifiable programs is complex
  - apparent in project management and maintenance problems

- positive tendencies
  - improved methodologies, tools and libraries
  - vanishing hardware limitations
  - increased qualification and experience
- negative tendencies
  - exponential functionality increase
  - new potentials: multimedia, networks, grid, ...

Catching Up With Hardware

"...as long as there were no machines,
programming was no problem at all;
when we had a few weak computers,
programming became a mild problem,
and now we have gigantic computers,
programming has become an equally gigantic problem."

Edsger W. Dijkstra, 1972 Turing Award Lecture

What is Good Software?

Goals & Solutions

- external criteria
  - what clients expect
- internal criteria
  - what to expect from your solution

- language techniques
  - language technology to use for meeting internal criteria
- design techniques
  - how to meet internal criteria with a given language technology

External Criteria

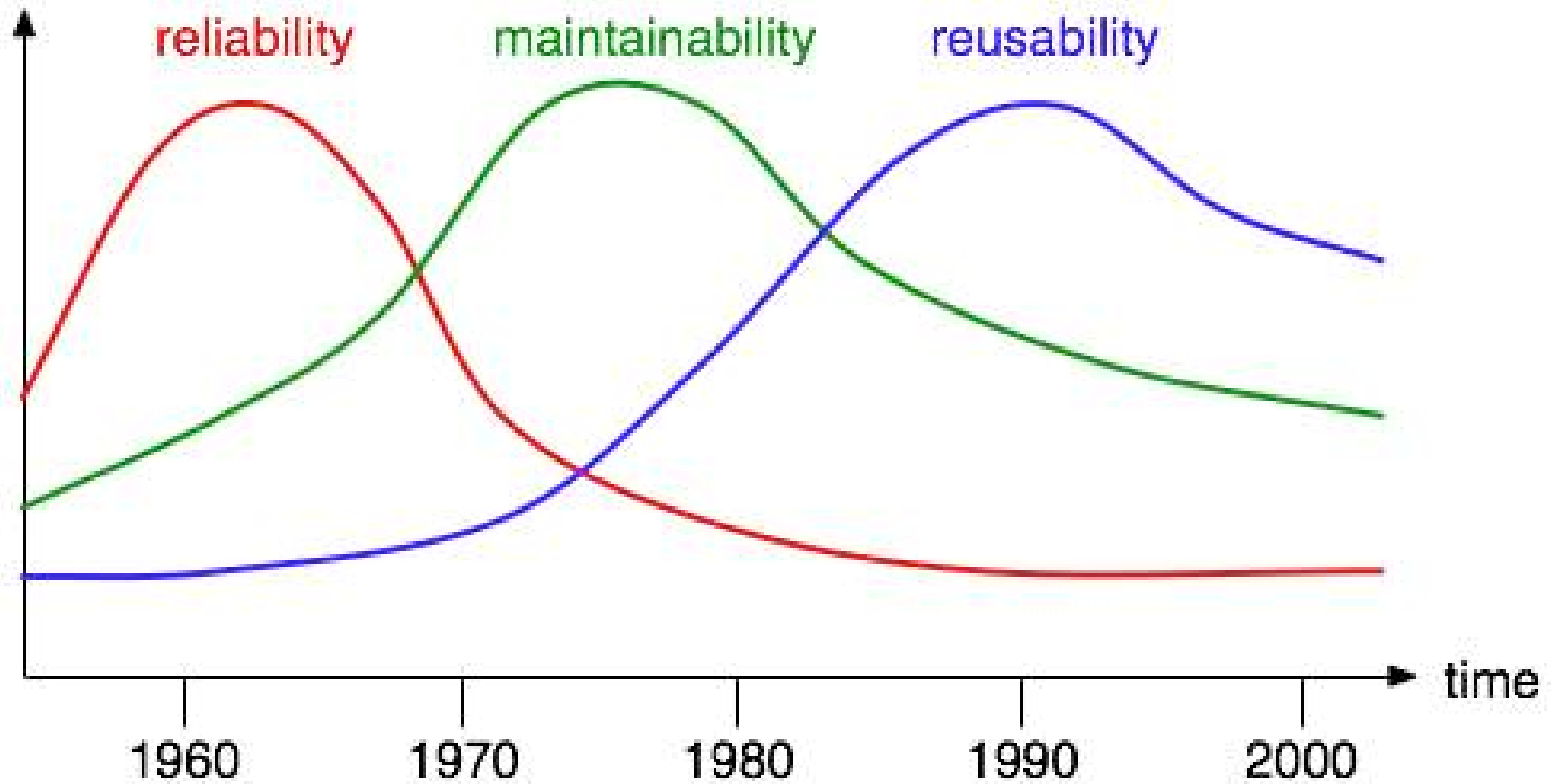
- correctness, robustness, efficiency, ...

more important in our context:

- maintainability
  - remove inadequacies
  - adapt to changing needs
  - extend functionality quickly
  - integrate with other systems
- reuse
  - use components in other projects

Software spends 60%-80% of its lifetime in maintenance!

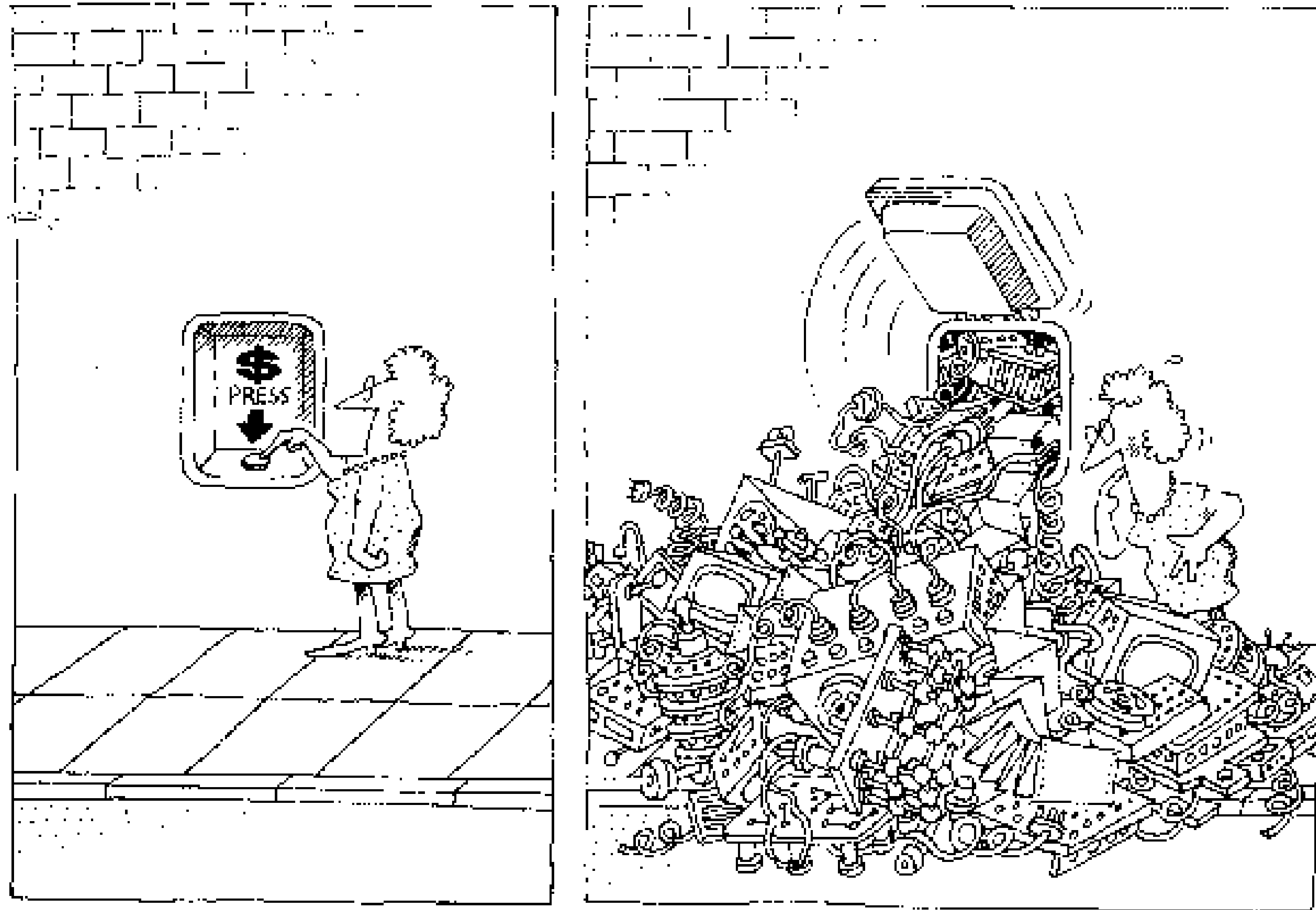
Change of Emphasis



Internal Criteria

- *what*, not *how*
  - demand *behaviour*, not a solution
- information hiding
  - system boundaries which stop change
- low coupling & high cohesion
  - create entities with a well-defined meaning
  - use interfaces to decouple entities
- separation of concerns
  - consider different concerns independently

Internal Criteria



The task of the software development team is to engineer the illusion of simplicity.

Very Early Days

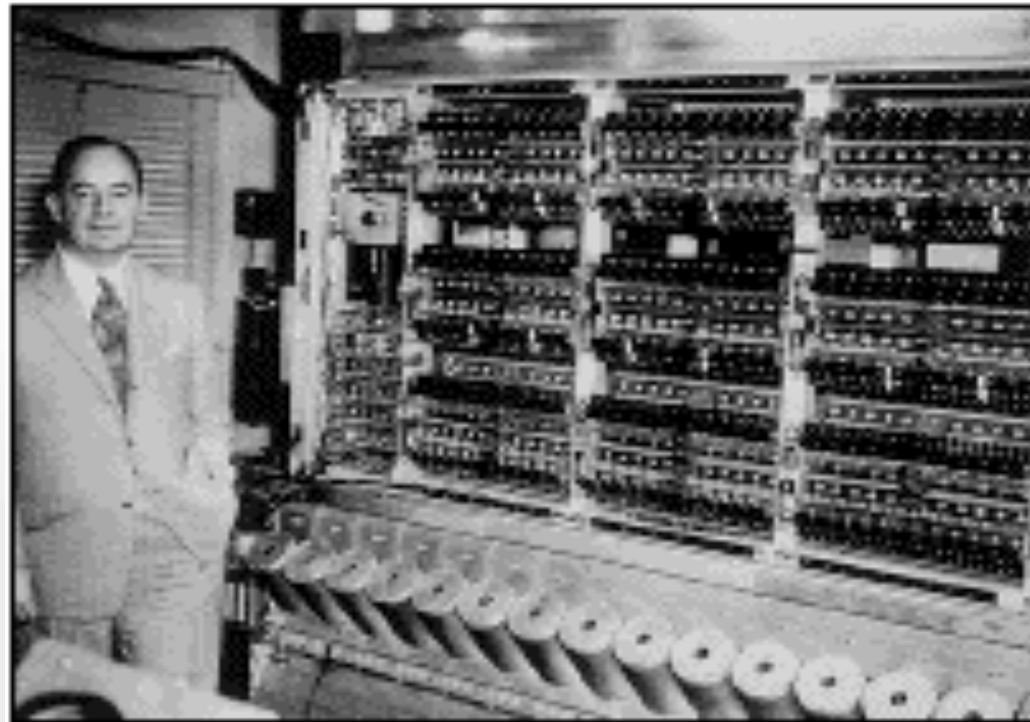
- programming language?
  - fixed set of operations
  - manual entering of input data
  - execution of chosen operation

Inflexible!

Hahn & Schuster, sample 1792

Quite Early Days

- von Neumann
  - operations are part of the input data
  - data and programs live in the same memory
  - self-modifying code was not unusual



Assembler

- abstracts (a little) from the bare machine
- symbols used for addresses and operations
  - locations automatically resolved
- no type information
  - does a location hold a number or a string?
- subroutines through macros only
  - minimal abstraction

Assembler

Labels for addresses

Mnemonics for instructions

Names for registers and memory cells

1. 156C
2. 166D
3. 5056
4. 306E
5. C000

machine language

```
START  LD    R5, PRICE
       LD    R6, TAX
       ADDI  R0, R5, R6
       ST    R0, TOTAL
END    HLT
```

assembly language

Assembler: the Value of Subroutines

- code reuse
  - duplication avoided (penalty for call/return)
  - enables recursion!
- code maintenance
  - fix a bug / adapt code once, for all callers

Language Genealogy

Plankalkül

imperative | object-oriented | functional | logic-based

1950

Fortran

Lisp

1960

Cobol

Algol 60

PL/1

Simula

Prolog

1970

Algol 68

Pascal

Smalltalk

C

ML

1980

Ada

C++

Miranda

Eiffel

1990

Haskell

Ada95

Java

2000

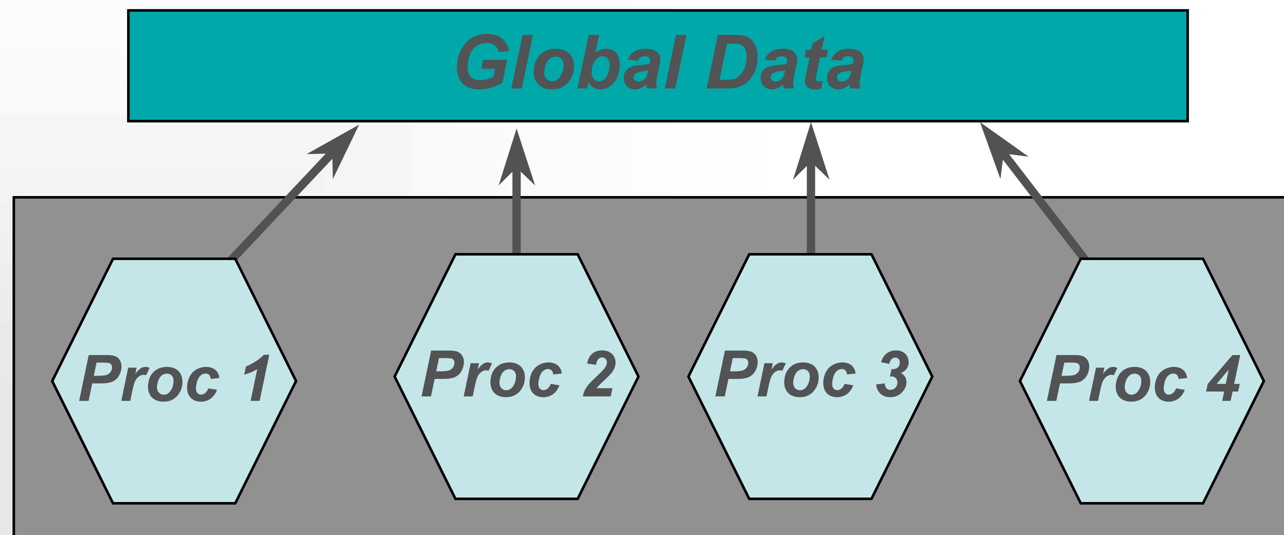
Language Generations

- first generation (1954-58)
  - mathematical expressions
  - Fortran I (Formula translation)
- second generation (1959-61)
  - subroutines, data types
  - Fortran II: subroutines, separate compilation
  - Algol 60: block structure, recursion, data types
  - Lisp: programs as data, garbage collection
- third generation (1962-70)
  - blocks, typing, classes
  - Pascal: simple successor to Algol 60
  - Simula: classes, data abstraction

Fortran

program "what" not "how"!

- milestone in computer science
  - high-level language with a compiler
- simple programming paradigm, though
  - global data with manipulating functions



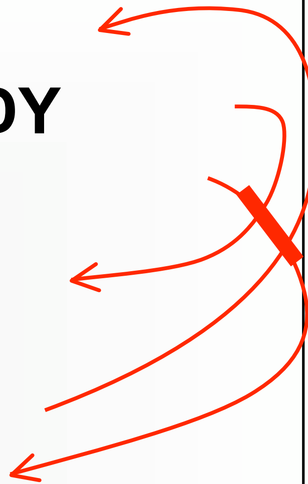
Aside: Program "what" not "how"...

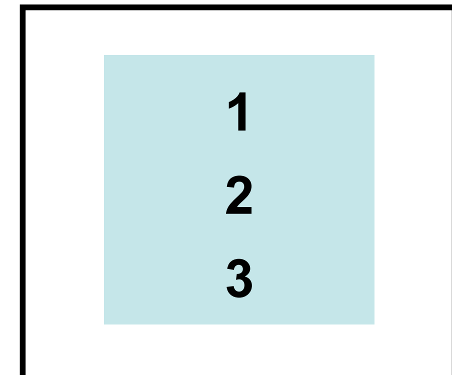
"goto considered harmful"

What does that thing do?

What happens at which time?

```
    i = 1
TEST: if i < 4
    then goto BODY
    else goto END

BODY: print(i)
    i = i + 1
    goto TEST
END:
```





Aside: Program "what" not "how"...

```
     i = 1
LOOP: print(i)
     i = i + 1
     if i < 4
       goto LOOP
```

Programming with Style

- **this is “elegant” code**
  - concise
  - nicely named
  - reflects the structure
  - idiomatic

We can actually only encourage style…

Aside: Program "what" not "how"...

Structured Programming ('64 –'75)

control structures in the language

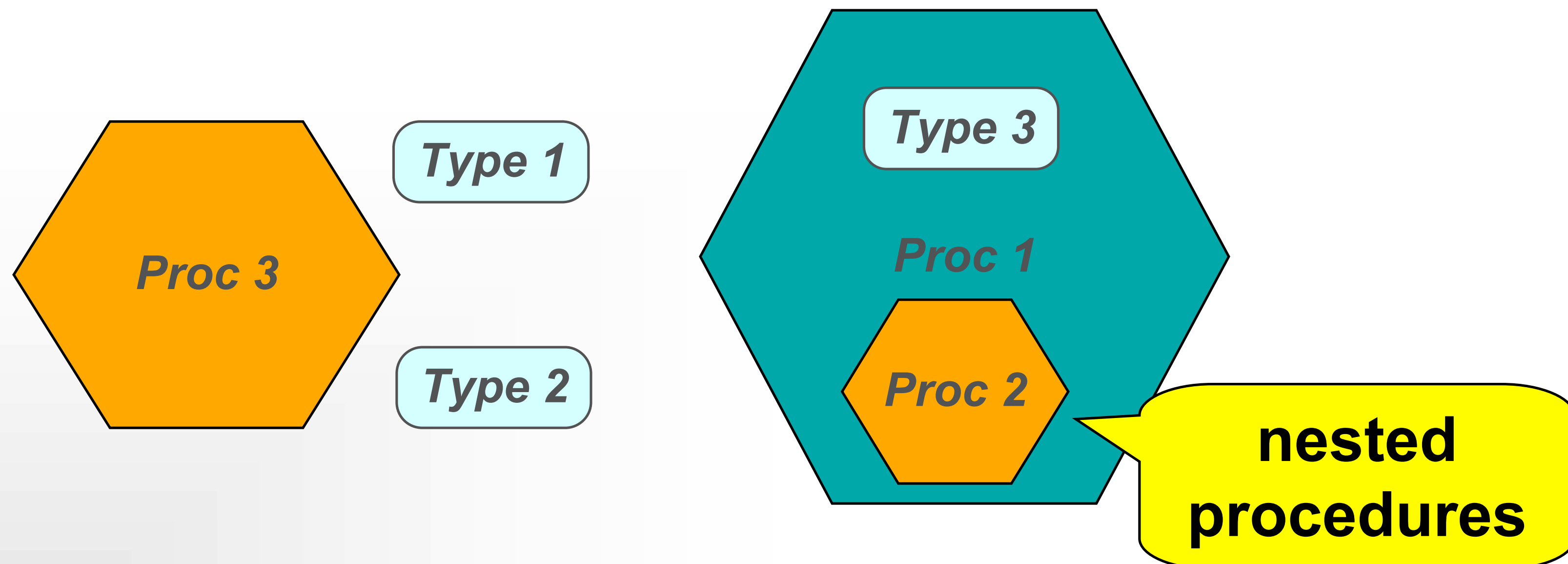
```
i = 1
while (i < 4) {
  print(i)
  i = i + 1
}
```

- **PL designer makes the rules**
  - new words, new grammar
- **you will write elegant code**
  - don't use words like "goto"
  - **say "while" when you "mean it"**

- **abstraction**: one "while-loop" instead of details of loop execution
- code is simpler to write, understand, debug

Pascal

- data types become first-class language entities
  - records with attributes (objects without methods)



Programs = Algorithms + Data Structures

N. Wirth

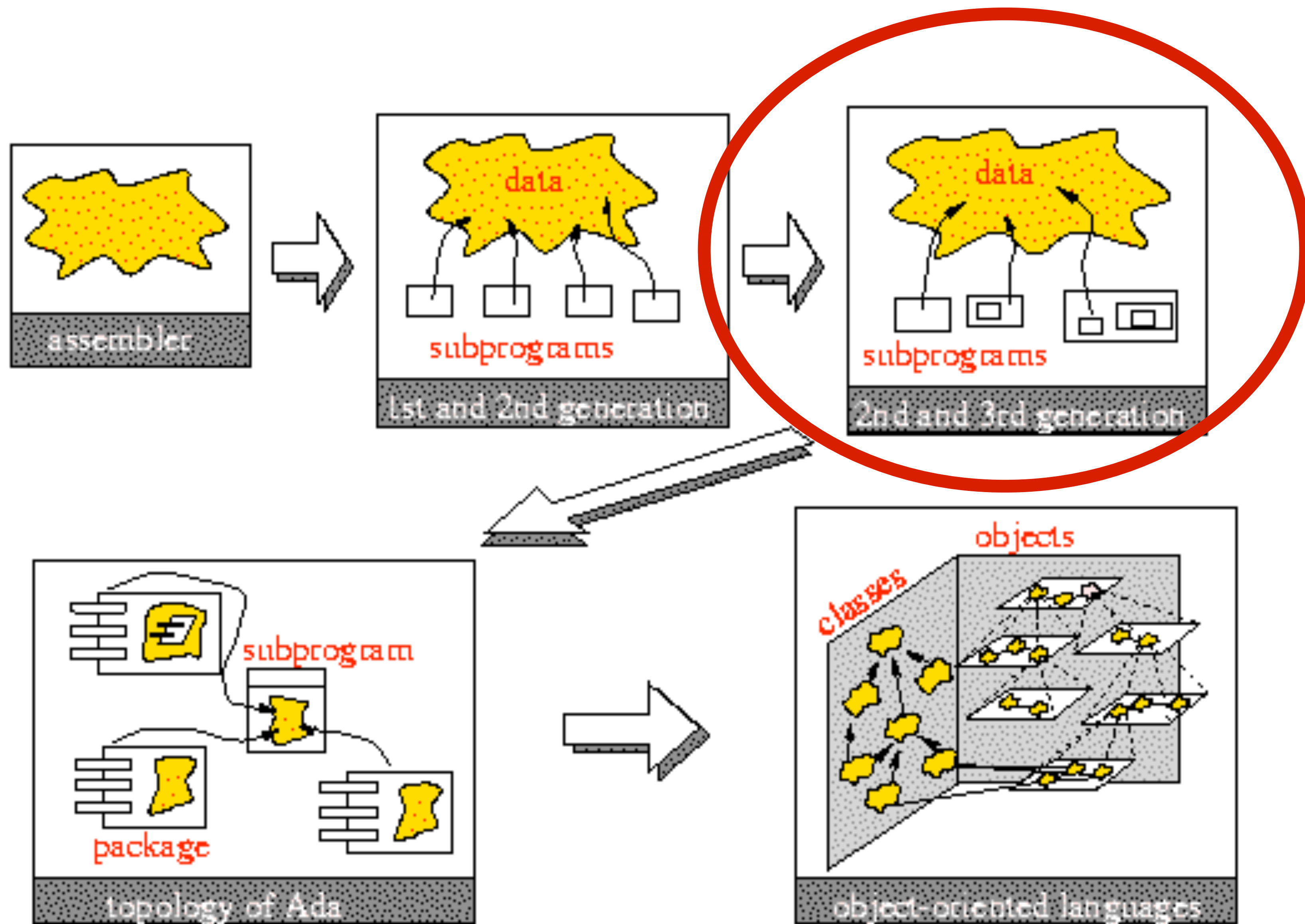
Pascal "Frequency"

```
Program frequency;
  const size: integer = 80;
var
  s: string[size];
  i: integer;
  c: char;
  f: array[1..26] of integer;
  k: integer;

begin
  writeln('Enter line');
  readln(s);

  for i := 1 to 26 do
    f[i] := 0;

k:=ord(s[0]);
for i := 1 to k do
  begin
    c := asLowerCase(s[i]);
    if isLetter(c) then
    begin
      k := ord(c) -
           ord('a') + 1;
      f[k] := f[k] + 1
    end
  end;

  for i := 1 to 26 do
    write(f[i], ' ')
end.
```

Structured Programming

- define tasks to be performed
  - break tasks into smaller and smaller pieces
  - until you reach an implementable size
- design how functions interact
  - what's the input / output
- define the data structures to be manipulated
- group functions into components
  - "units" or "modules"

Language Models



Airline Reservation System

– Enquiry on Flights –

Flight sought from: *Santa Barbara* To: *Paris*

Departure on or after: *21 Nov* On or before: *22 Nov*

Preferred airline (s):

Special requirements:

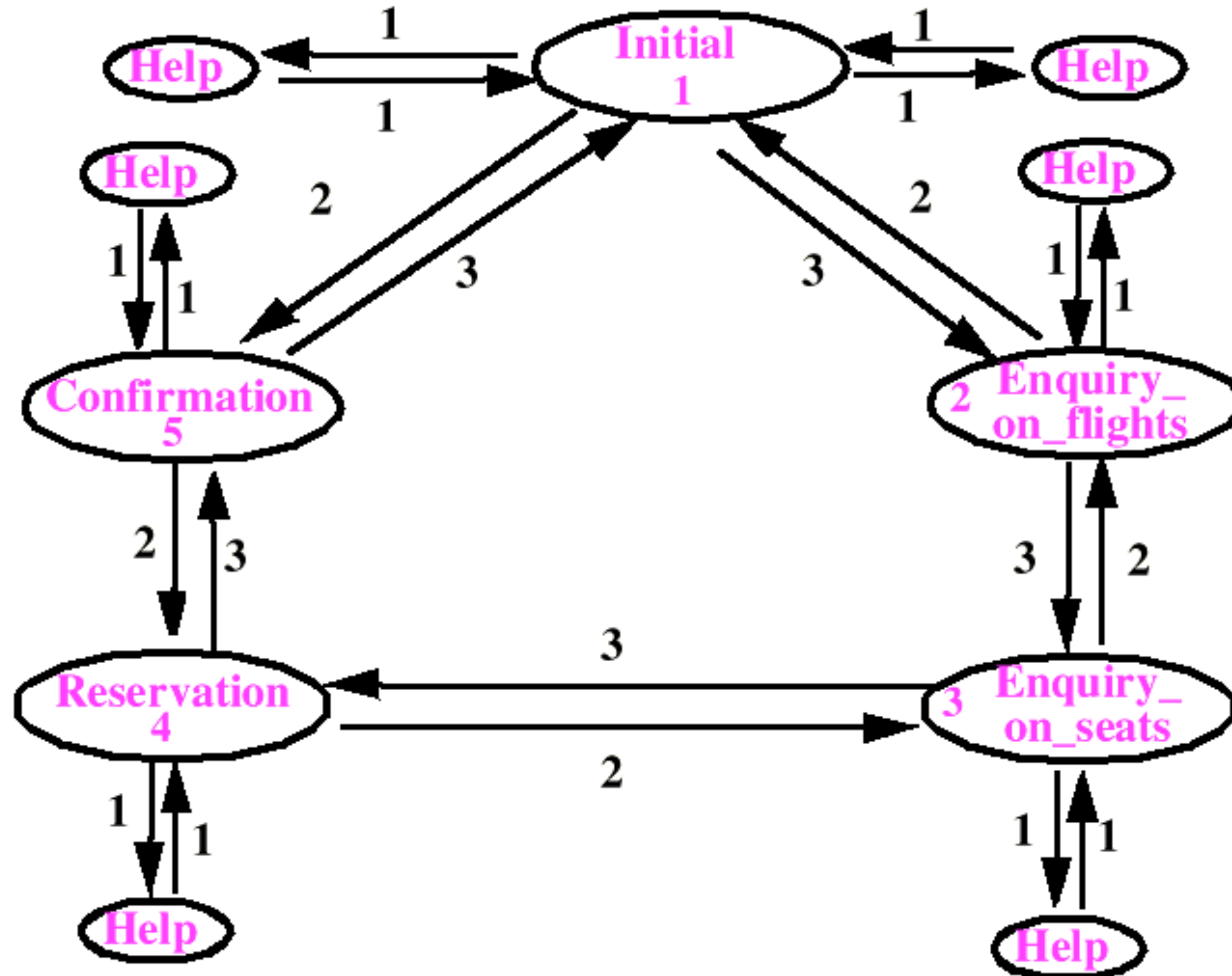
AVAILABLE FLIGHTS: **1**

Flt# AA 42 **Dep 8:25** **Arr 7:45** **Thru: Chicago**

Choose next action:

0 — **Exit**
1 — **Help**
2 — **Further enquiry**
3 — **Reserve a seat**

Airline Reservation System



A Case of "Spaghetti Code"

EnquiryOnFlights:

"Display *Enquiry on flights* panel"

"Read user's answers and choice *C* for the next step"

case *C* in

0 : goto *Exit,*

1 : goto *Help*,

2 : goto *Reservation*,

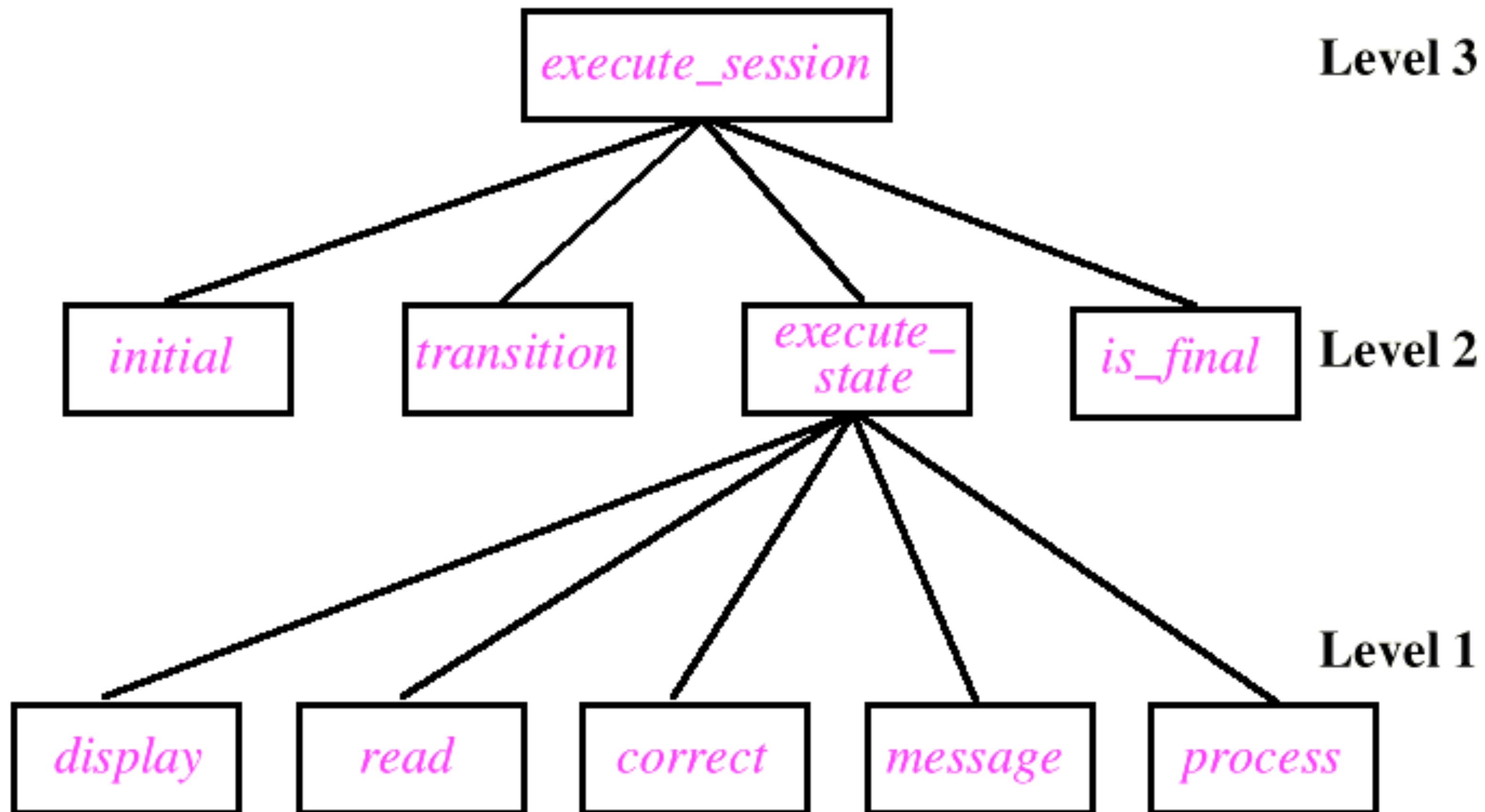
end case

...

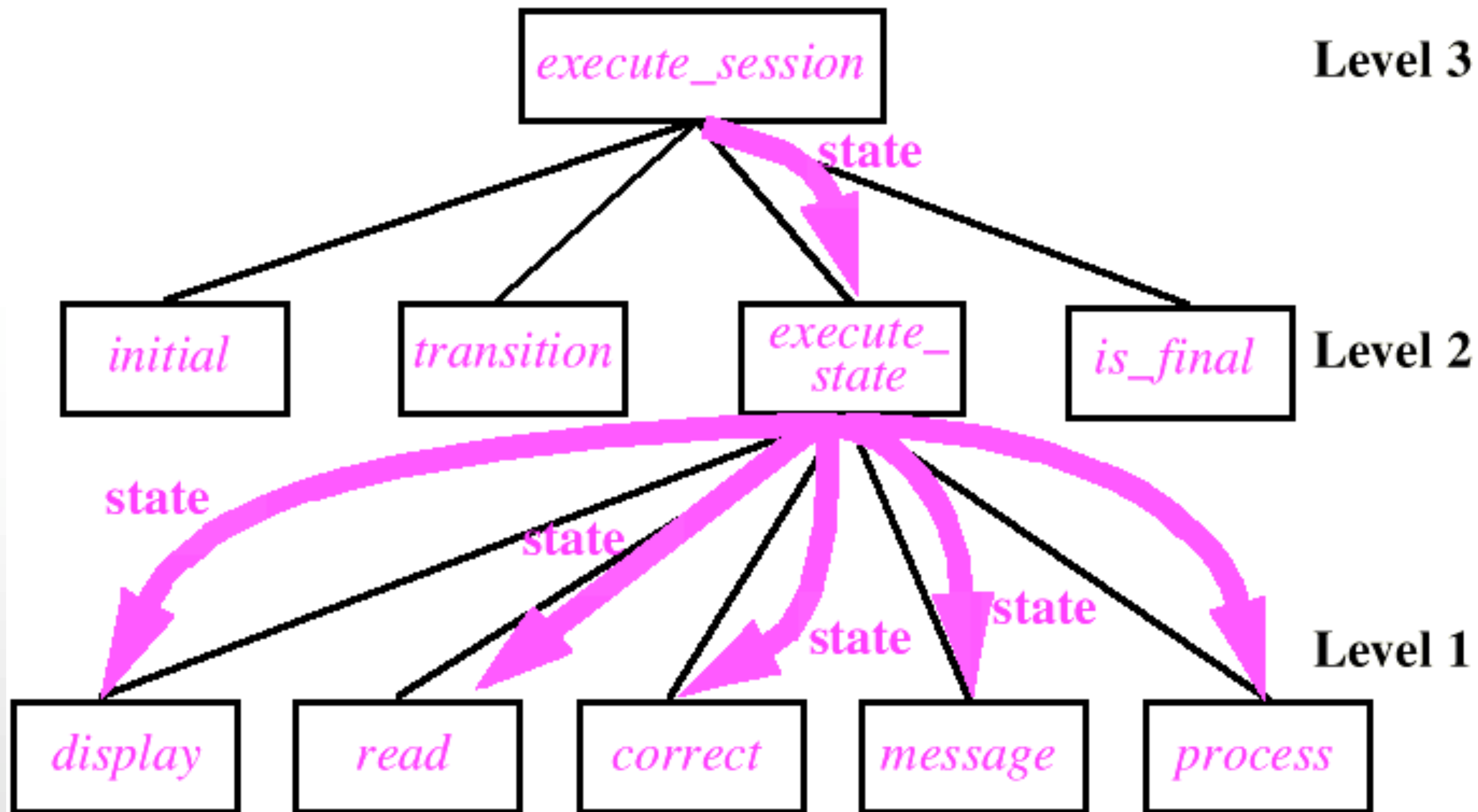
Topology of interface masks is *hardwired* into the system.

Adding a state or changing a transition results in changes to the system's central control structure.

Functional Decomposition



Passing Data



Passing Data

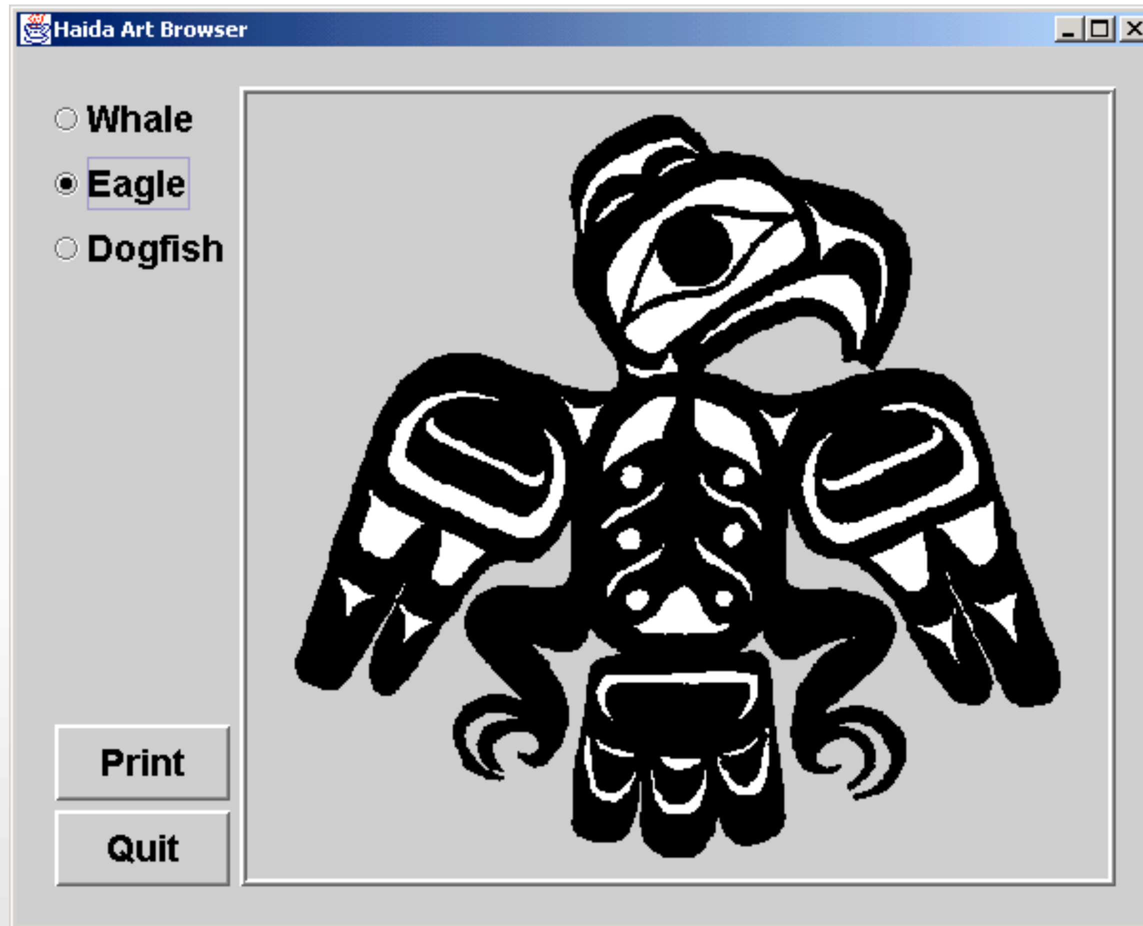
execute_state (**in** *s*: *STATE*; **out** *c*: *CHOICE*)
display (**in** *s*: *STATE*)
read (**in** *s*: *STATE*; **out** *a*: *ANSWER*)
correct (**in** *s*: *STATE*; *a*: *ANSWER*): *BOOLEAN*
message (**in** *s*: *STATE*; *a*: *ANSWER*)
process (**in** *s*: *STATE*; *a*: *ANSWER*)

State is passed around

```
execute_state (in s : State; out : Choice)
...
  if (s==flightEnquiry)
     ...
  if (s==flightReservation)
     ...
```

Aside: Program "what" not "how"...

Implementation of a simple browser...



… image browser
with structured programming

Aside: Program "what" not "how"...

Browser: implementation

```
main () {
  draw_label("Art Browser");
  m = radio_menu(
      {"Whale", "Eagle", "Dogfish"});
  q = button_menu({"Quit"});
  while ( ! check_buttons(q) ) {
    n = check_buttons(m);
    draw_image(n);
  }
}

draw_label (string) {
  w = calculate_width(string);
  print(string, WINDOW_PORT);
  set_x(get_x() + w);
}

radio_menu(labels) {
  i = 0;
  while (i < labels.size) {
    radio_button(i);
    draw_label(labels[i]);
    set_y(get_y() + RADIO_BUTTON_H);
    i++;
  }
}
```

```
radio_button (n) {
  draw_circle(get_x(), get_y(), 3);
}

draw_circle (x, y, r) {
  %%primitive_oval(x, y, 1, r);
}

button_menu(labels) {
  i = 0;
  while (i < labels.size) {
    draw_label(labels[i]);
    set_y(get_y() + BUTTON_H);
    i++;
  }
}

draw_image (img) {
  w = img.width;
  h = img.height;
  do (r = 0; r < h; r++)
    do (c = 0; c < w; c++)
      WINDOW[r][c] = img[r][c];
}
```

Aside: Program "what" not "how"...

Better Style: Separation of Concerns

```
haida_browser:
  main () {
   draw_label("Haida Art Browser");
   m = radio_menu(
       {"Whale", "Eagle", "Dogfish"});
   q = button_menu({"Quit"});
   while ( ! check_buttons(q) ) {
     n = check_buttons(m);
     draw_image(n);
   }
  }
```

```
menus:
  radio_menu(labels)
  button_menu(labels)
  check_buttons(menu)
```

```
graphics:
  draw_image(img)
  draw_label(string)
  draw_circle(x, y, r)
```

- group by:
  - "shared secrets"
  - common functionality
  - no upcalls
- modules with
  - clear, narrow interfaces

Aside: Program "what" not "how"...

Modular Programming

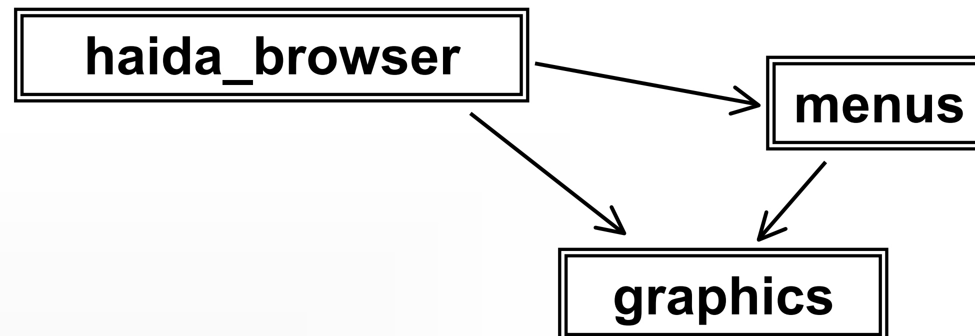
Module constructs in the language

- clear module interfaces
  - explicit
  - enforced
    - only through front door
    - can’t "read the mail"
  - type systems
  - opacity

```
module menus {
  exports:
    radio_menu(labels)
    button_menu(labels)
    check_buttons(menu)
}
```

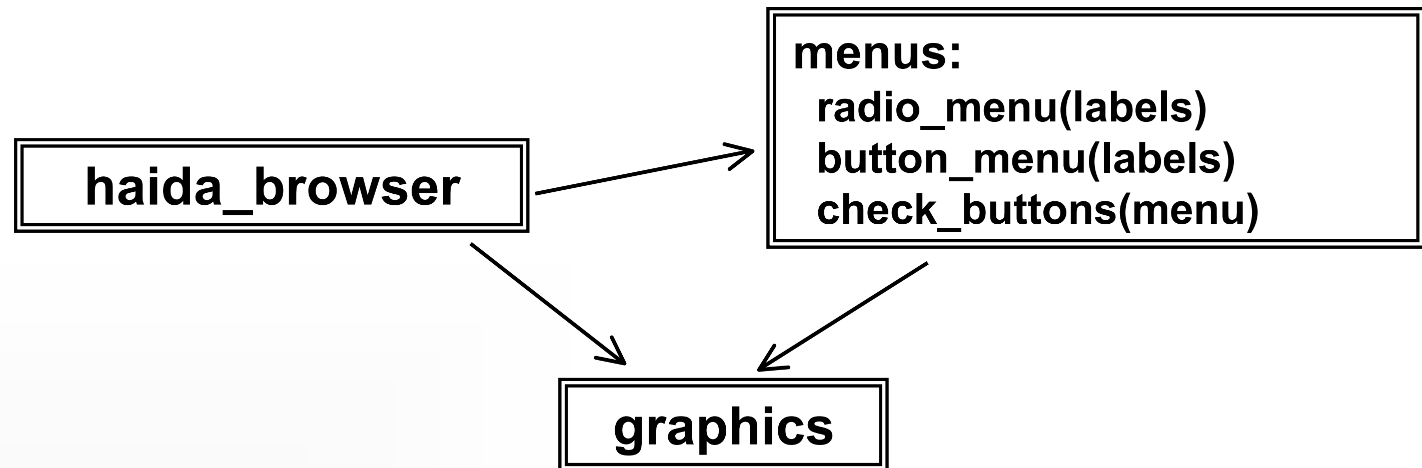
- "abstraction" mechanisms
  - from the outside, only the module abstraction is visible

Thinking Architecturally



- abstraction lets us
  - look at overall system structure

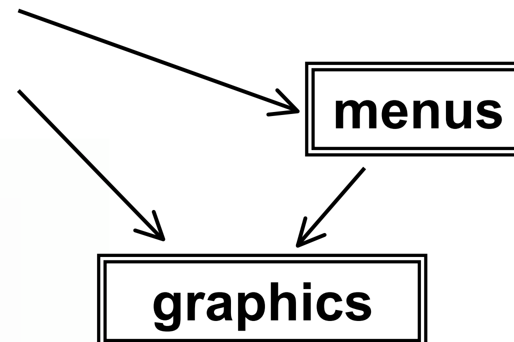
Thinking Architecturally



- abstraction lets us
  - look at overall system structure
  - zoom in on one part of the system at a time

Thinking Architecturally

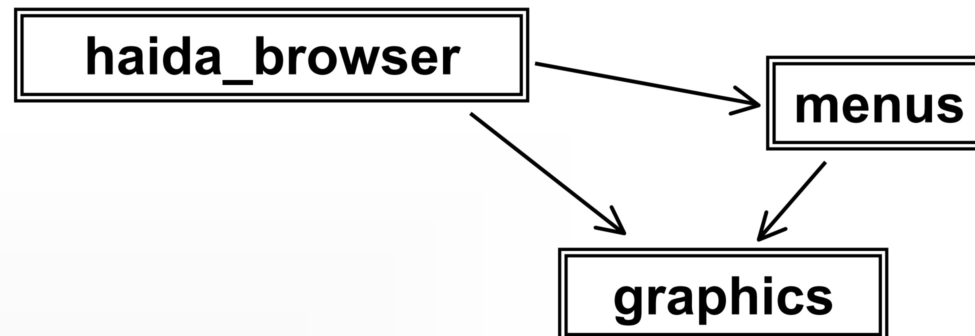
```
haida_browser:
  main () {
   draw_label("Haida Browser");
   m = radio_menu(
       {"Whale", "Eagle", "Dogfish"});
   q = button_menu({"Quit"});
   while ( ! check_buttons(q) ) {
     n = check_buttons(m);
     draw_image(n);
   }
 }
```



- abstraction lets us
  - look at overall system structure
  - zoom in on one part of the system at a time
    - more or less detail

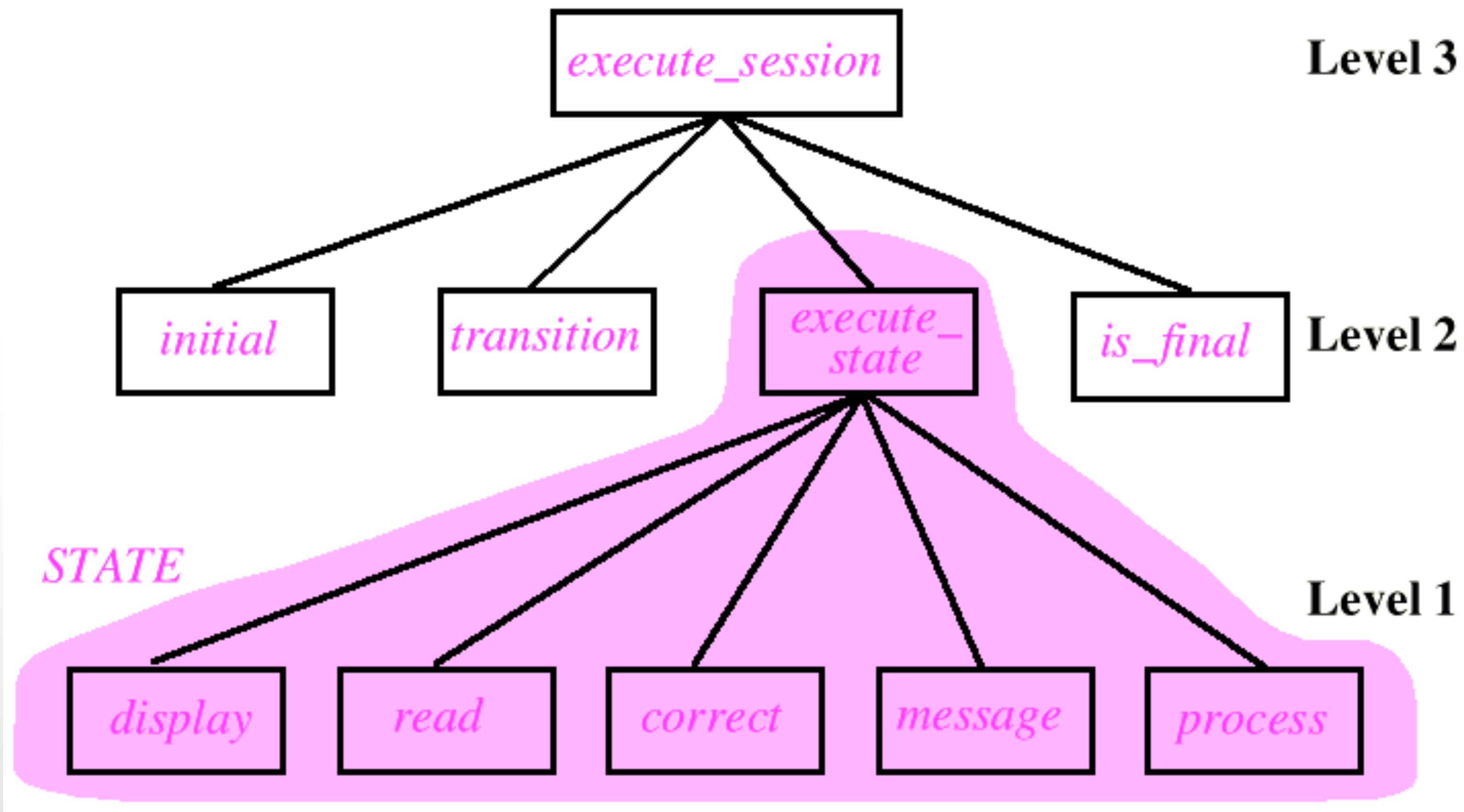
"Modular Programming"

- separation of concerns
  - organise code according to common functionality
  - PL mechanisms
    - enforce secrecy of module internals



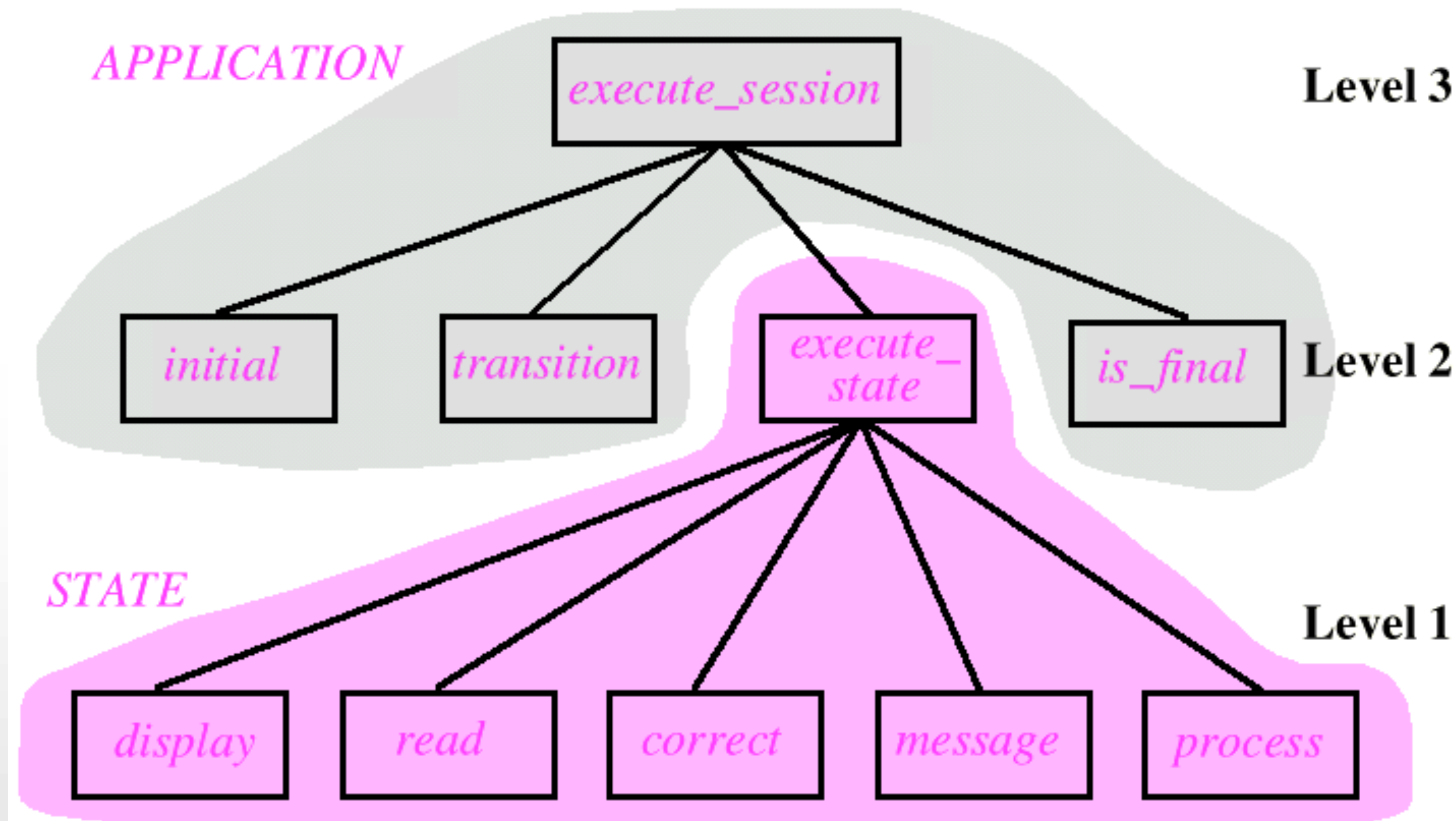
- abstraction
  - enables architectural thinking
  - treat the module as its interface

Identifying Objects



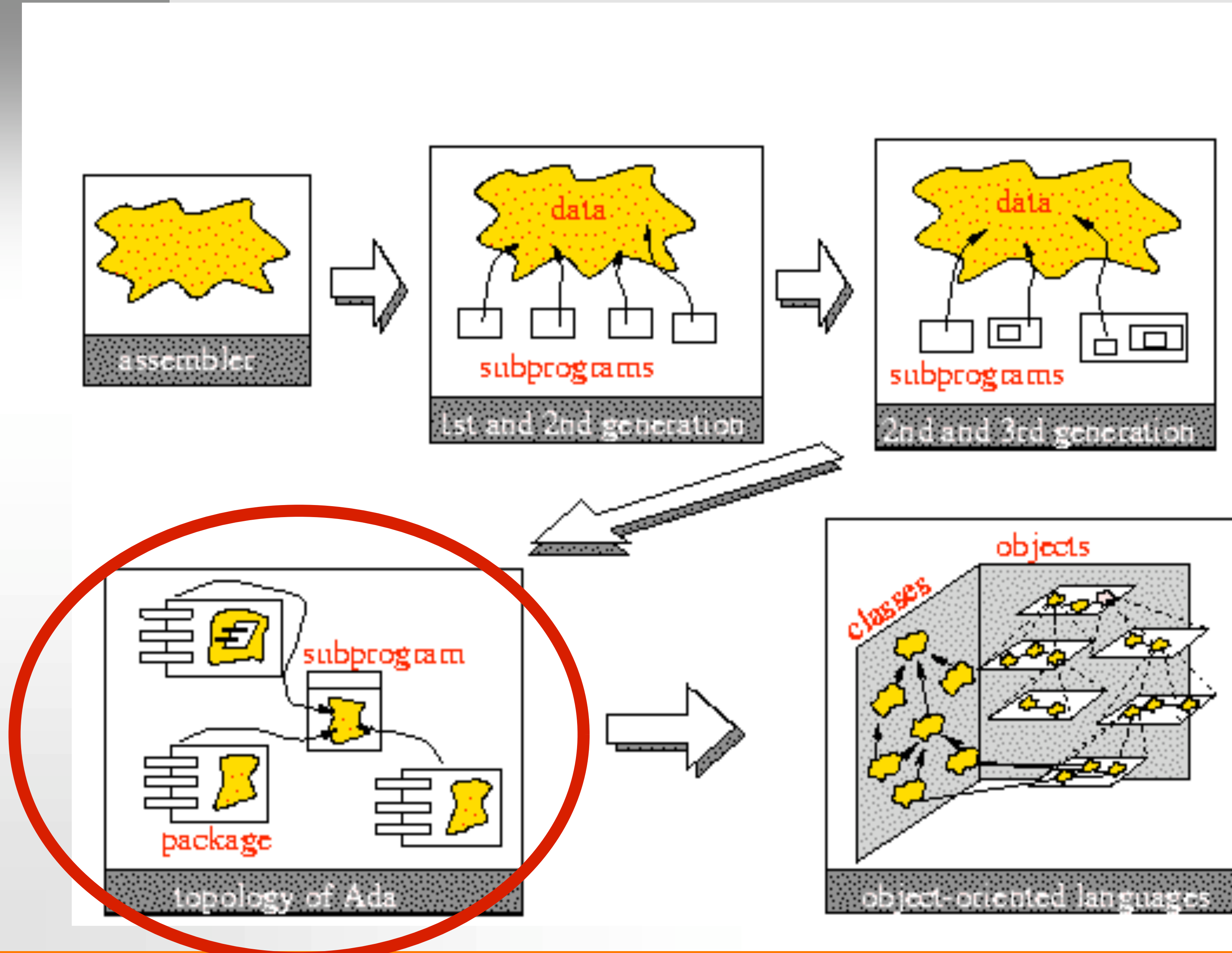
- turn functions on state into "State" with functions

Identifying Objects



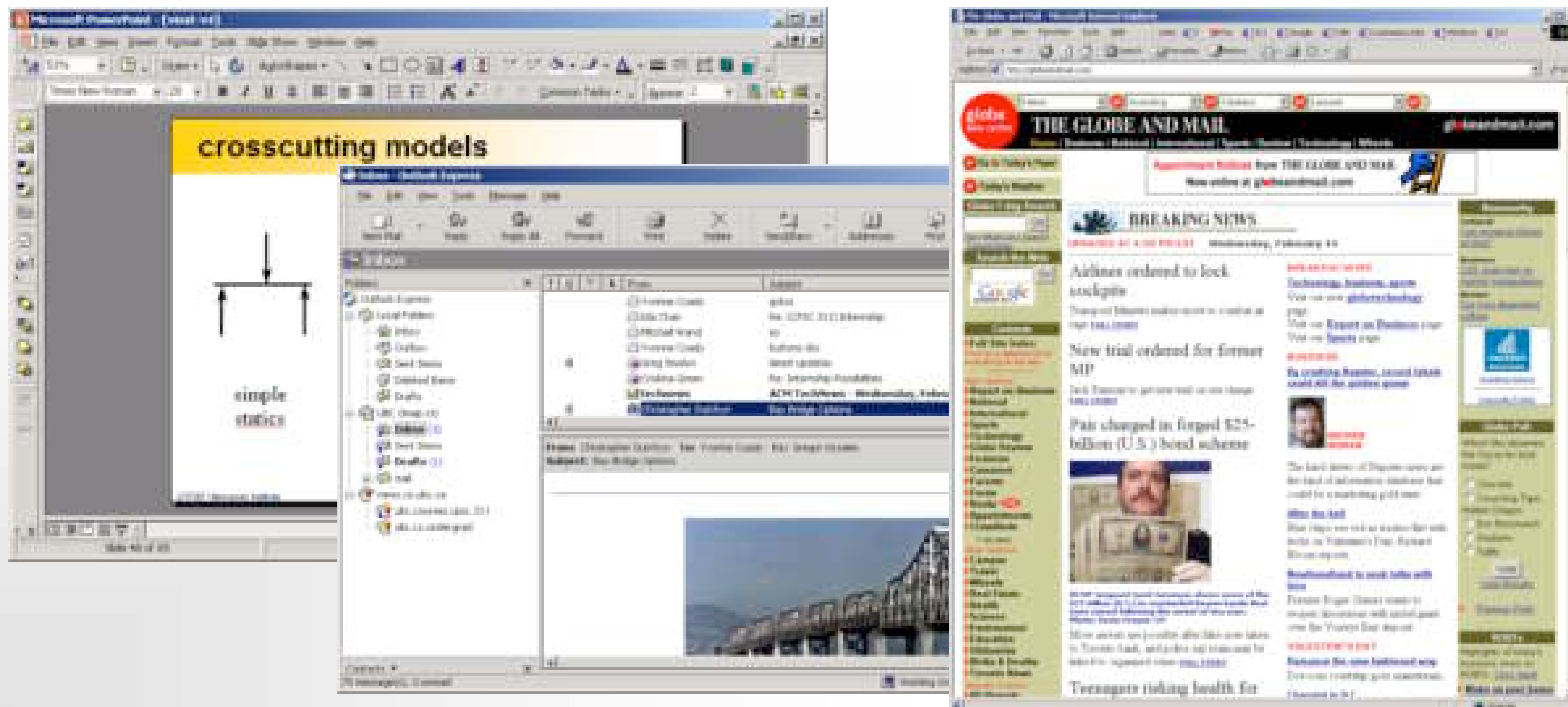
- add "Application" to replace top level function

Language Models



Systems of Applications

- Powerpoint, Outlook, browsers…
- tangling problems arise again



Variations of Functionality

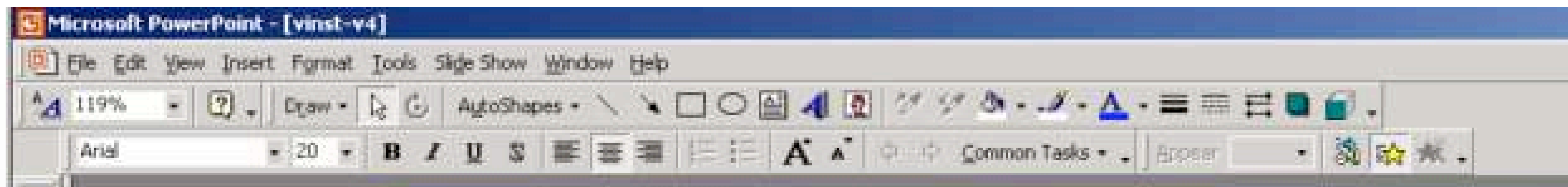
pull-down menus

toolbars

toolbar buttons

combo: text box and list

color choosers



even with modular programming
this appears to be incredibly complex…

Coping with Complexity

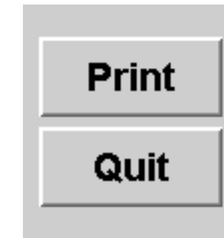
- Simula [Dahl & Nygaard 64, 68]
- Smalltalk [Alan Kay 70 – 80]

- we cope with complexity all the time
  - world has a diversity of complex objects
  - abstraction, classification and specialisation

- “model” programs after the world

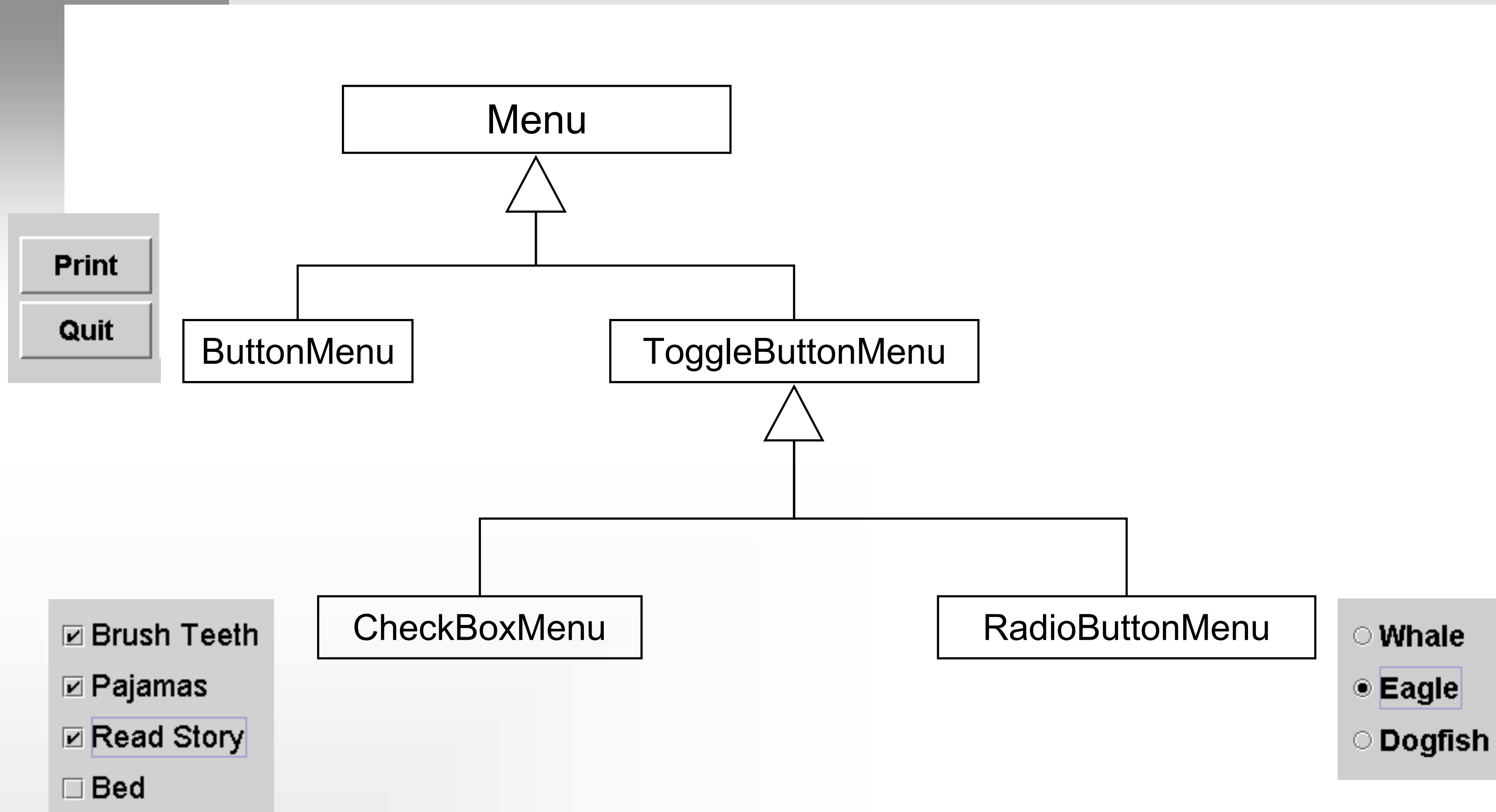
"Object-Oriented" Programming

- aka OOP
- “objects” are…
  - animate
    - know how to do things
    - code plus data
    - little computers waiting to be told what to do
  - classified
    - *specialisations* of other objects

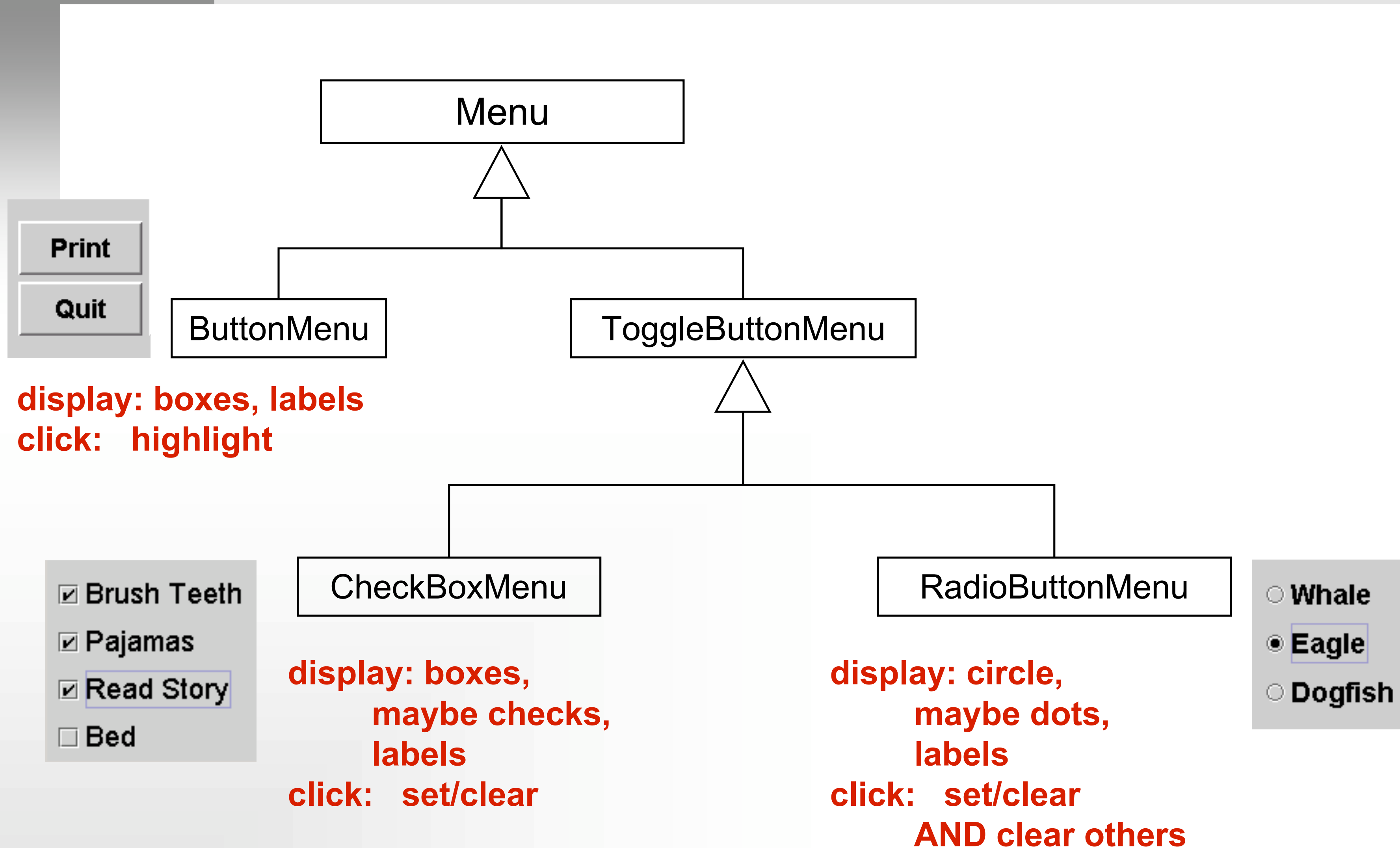


- a “class” is a cookie cutter for objects
  - *specialisations* of other classes

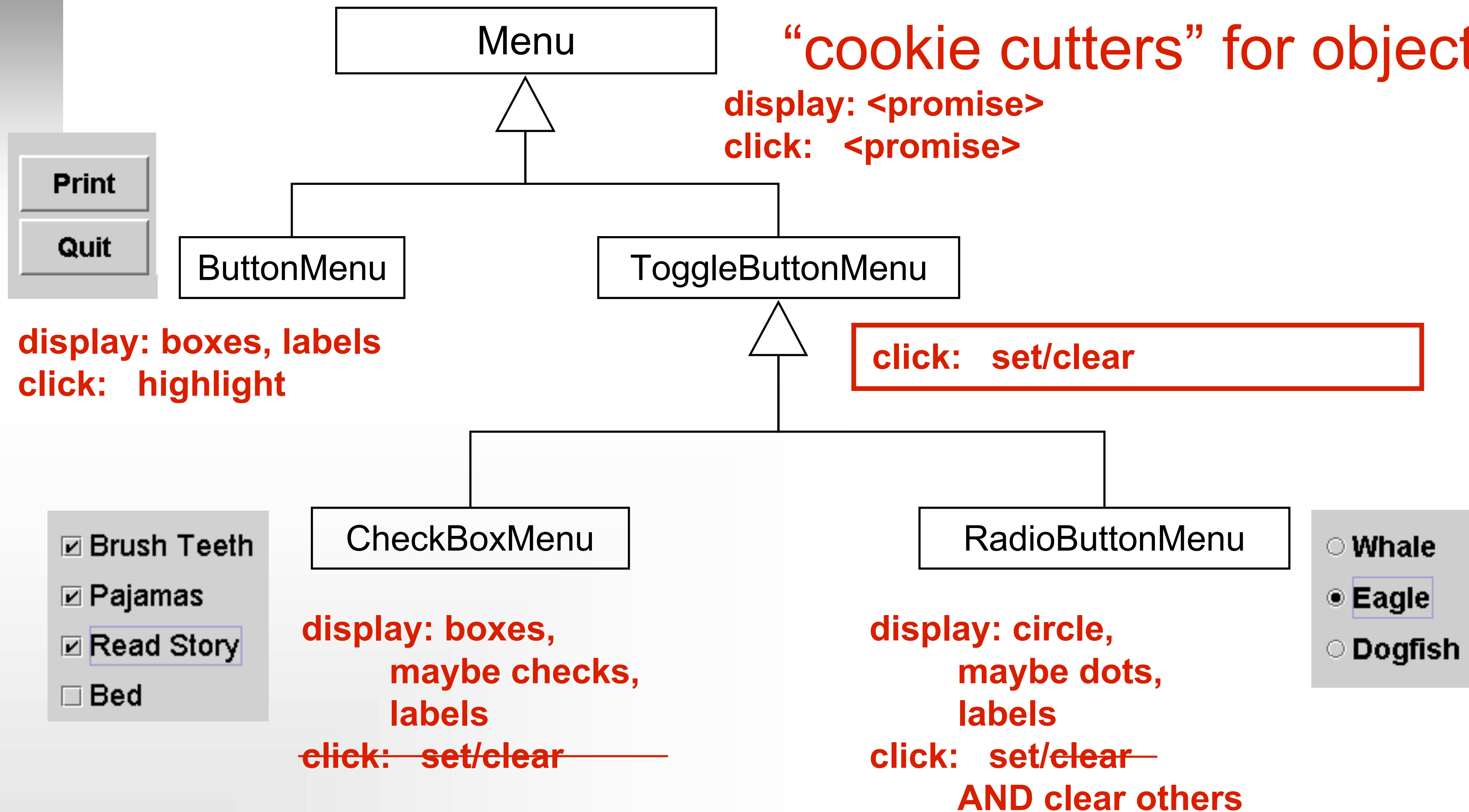
A Class Hierarchy



Operations on Objects



Inheritance



PL Support

```
class RadioButtonMenu
    extends ToggleButtonMenu {
  void display() {
    i = 0;
    while (i < buttons.length) {
      drawCircle();
      drawLabel(buttons[i]);
      if ( buttons[i].isSet() )
        drawHighlight(i);
    }
  }

  void click(i) {
    super(i);
    clearOthers(i);
  }
}
```

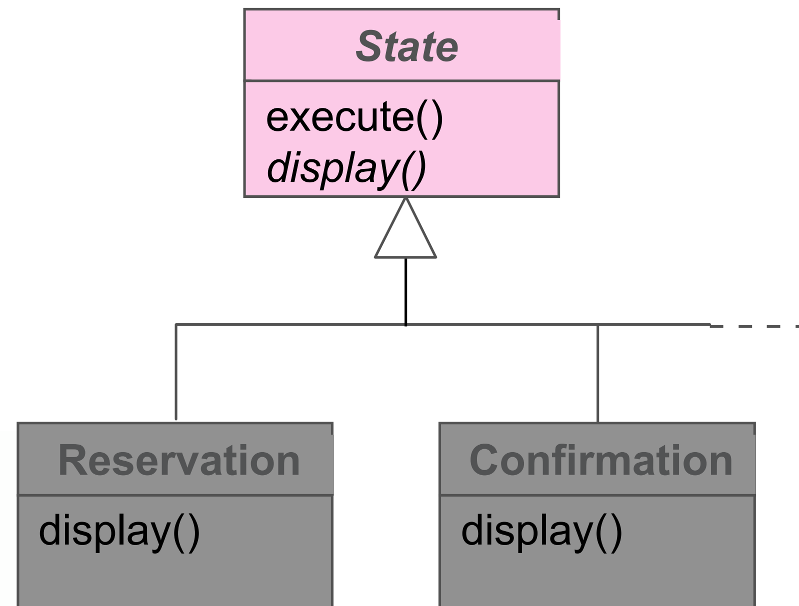
a “subclass” of ToggleButton

defines two operations

this operation builds on the superclass’s version of it

State Object

```
class STATE feature
  ...
  execute(out c : Choice)
  display()
  read
  ...
end
```



"STATE" argument disappears from signatures

Simula

- simulation programming language from Norway, 1966
- *activities* which can be instantiated as *processes*
- processes have their own data and behaviour
- inheritance through "block prefixes"

Smalltalk

- the LISP of OO languages
  - extremely little syntax
  - highly reflective
- virtual machine
  - true *write once, run everywhere*
- dynamic typing
  - type annotations unnecessary
  - unrestricted polymorphism

Smalltalk

All of Smalltalk's keywords

true	*the value for truth*
false	*the value for falsity*
nil	*the value for undefined references*
self	*referring to the receiver of a message*
super	*redirecting message lookup to parents*

Smalltalk

Java external iteration

```
Enumeration e=names.elements();

while (e.hasMoreElements())
  System.out.println(e.nextElement());
```

- active consumption of a structure

Smalltalk internal iteration

```
names do: [ :name | name printOn: Transcript ]
```

- passing code to be applied to the structure *to* the structure

OO "Frequency"

```
| s freqs |

s := Prompter prompt: 'Enter line' default: ''.

freqs := Bag new.
s do: [ :c | c isLetter ifTrue:
                 [freqs add: c asLowerCase]
   ].
^freqs

"output frequencies"
Transcript show: (freqs sortedCounts).
```

- proper choice of data structure does half the work
- rich libraries yield short programs