Software Engineering

Third class

Lecture 11

COMPONENT-LEVEL DESIGN

Design model must be translated into operational software. To accomplish this, the design must be represented at a level of abstraction that is close to code. Component level design (*procedural design*) establishes the algorithmic detail required to manipulate data structures, effect communication between software components via their interfaces, and implement the processing algorithms allocated to each component.

11.1 Why Component-level design is important?

You have to be able to determine whether the program will work before you build it. The component-level design represents the software in a way that allows you to review the details of the design for correctness and consistency with earlier design representations (i.e., the data, architectural, and interface designs). It provides a means for assessing whether data structures, interfaces, and algorithms will work.

11.2 Structured Programming

The foundations of component-level design were formed in the early 1960s and were solidified with the work of Edsgar Dijkstra and his colleagues.

In the late 1960s, Dijkstra and others proposed the use of a set of constrained logical constructs from which any program could be formed. The constructs emphasized "maintenance of functional domain." That is, each construct had a predictable logical structure, was entered at the top and exited at the bottom, enabling a reader to follow procedural flow more easily.

The constructs are sequence, condition, and repetition. *Sequence* implements processing steps that are essential in the specification of any algorithm. *Condition* provides the facility for selected processing based on some logical occurrence, and *repetition* allows for looping. These three constructs are fundamental to *structured programming*—an important component-level design technique.

The structured constructs were proposed to limit the procedural design of software to a small number of predictable operations. Complexity metrics indicate that the use of

the structured constructs reduces program complexity and thereby enhances readability, testability, and maintainability. The use of a limited number of logical constructs also contributes to a human understanding process that psychologists call *chunking.*

The structured constructs are logical chunks that allow a reader to recognize procedural elements of a module, rather than reading the design or code line by line. Understanding is enhanced when readily recognizable logical patterns are encountered.

Any program, regardless of application area or technical complexity, can be designed and implemented using only the three structured constructs.

11.2.1 Graphical Design Notation

A) flowchart

A flowchart is quite simple pictorially. A box is used to indicate a processing step. A diamond represents a logical condition, and arrows show the flow of control. Figure 1 illustrates three structured constructs.

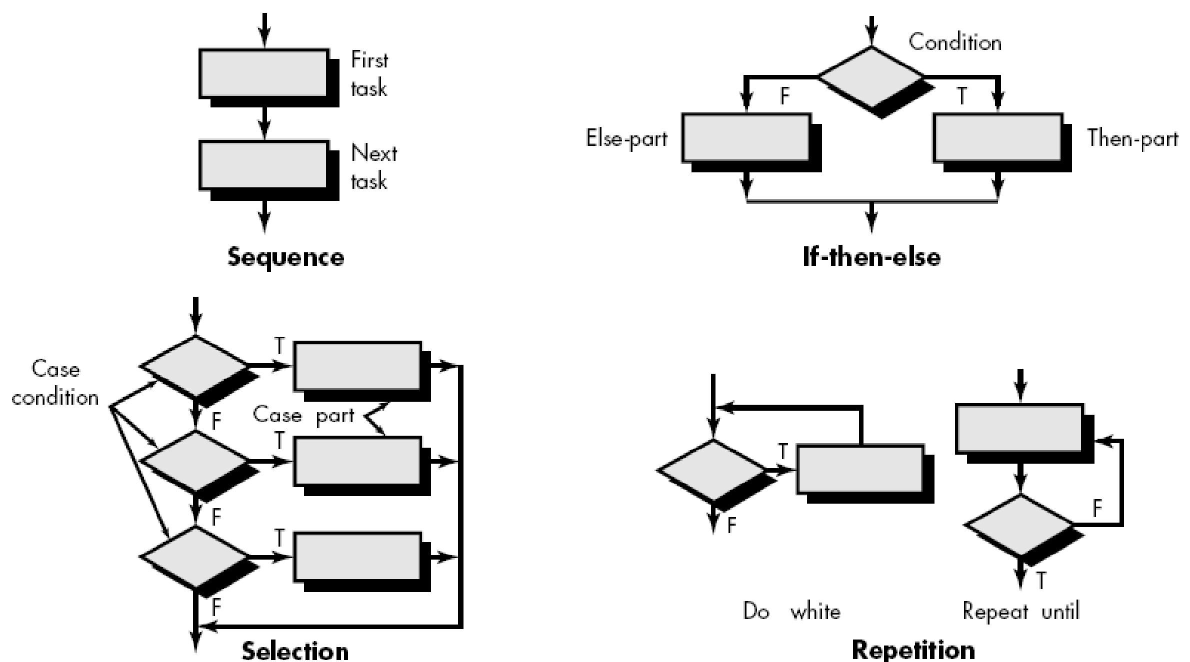

Figure 1: Flowchart constructs

The *sequence* is represented as two processing boxes connected by a line (arrow) of control. *Condition,* also called *if-then- else,* is depicted as a decision diamond that if

true, causes *then-part* processing to occur, and if false, invokes *else-part* processing. *Repetition* is represented using two slightly different forms. The *do while* tests a condition and executes a loop task repetitively as long as the condition holds true. A *repeat until* executes the loop task first, then tests a condition and repeats the task until the condition fails. The selection (or select-case) construct shown in the figure is actually an extension of the if-then-else. A parameter is tested by successive decisions until a true condition occurs and a case part processing path is executed.

The structured constructs may be nested within one another as shown in Figure 2.

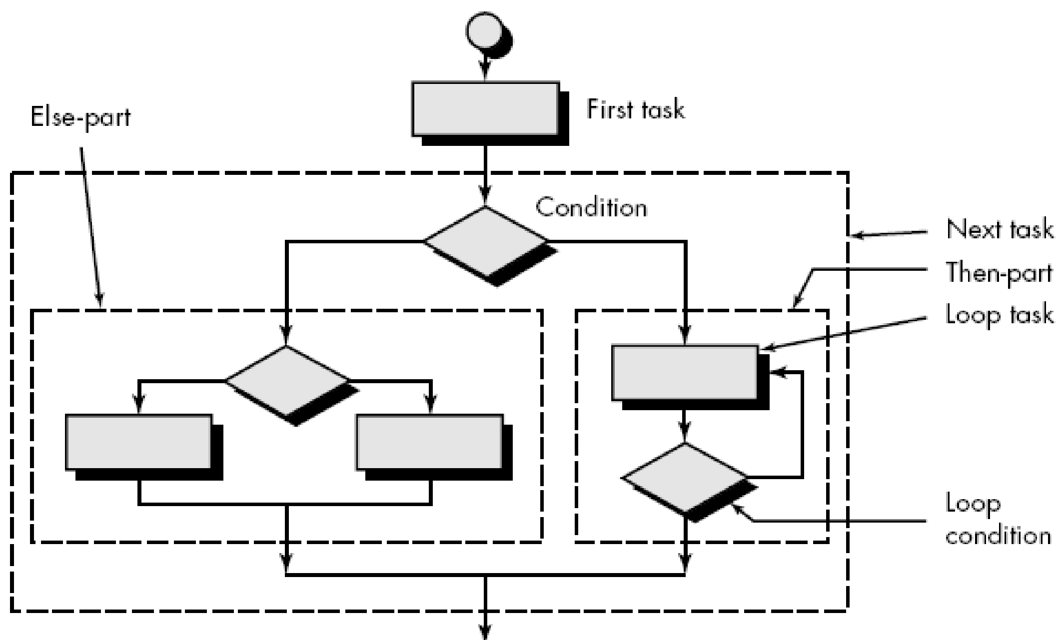

Figure 2 Nesting constructs

Referring to the figure, repeat-until forms the then part of if-then-else (shown enclosed by the outer dashed boundary). Another if-then-else forms the else part of the larger condition. Finally, the condition itself becomes a second block in a sequence.

B) Box Diagram

Another graphical design tool, the *box diagram,* evolved from a desire to develop a procedural design representation that would not allow violation of the structured constructs. Developed by Nassi and Shneiderman and extended by Chapin (also called *Nassi-Shneiderman charts, N-S charts,* or *Chapin charts*).

The graphical representation of structured constructs using the box diagram is illustrated in Figure 3.

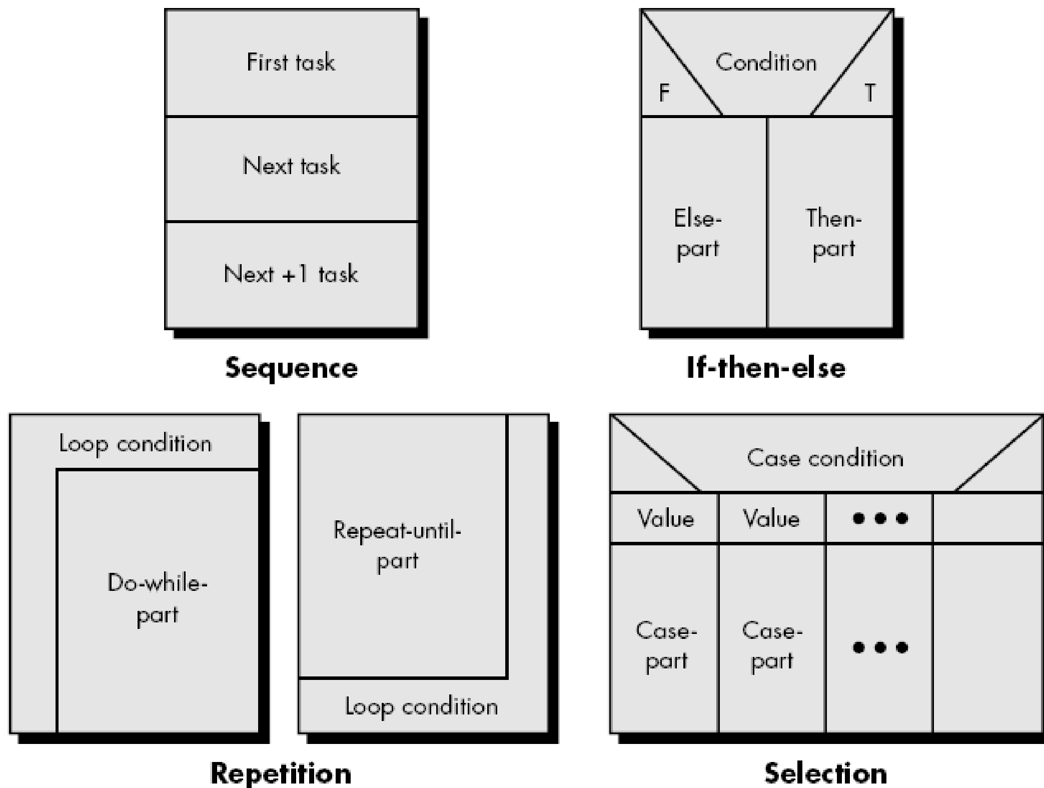

Figure 3 Box diagram constructs

The fundamental element of the diagram is a box. To represent sequence, two boxes are connected bottom to top. To represent if-then-else, a condition box is followed by a then-part and else-part box. Repetition is depicted with a bounding pattern that encloses the process (do-while part or repeat-until part) to be repeated. Finally, selection is represented using the graphical form shown at the bottom of the figure.

11.2.2 Tabular Design Notation

In many software applications, a module may be required to evaluate a complex combination of conditions and select appropriate actions based on these conditions. Decision tables provide a notation that translates actions and conditions (described in a processing narrative) into a tabular form. The table is difficult to misinterpret and may even be used as a machine readable input to a table driven algorithm.

Decision table organization is illustrated in Figure 4. Referring to the figure, the table is divided into four sections. The upper left-hand quadrant contains a list of all conditions. The lower left-hand quadrant contains a list of all actions that are possible based on combinations of conditions. The right-hand quadrants form a matrix

indicates condition combinations and the corresponding actions that will occur for a specific combination. Therefore, each column of the matrix may be interpreted as a processing *rule.*

	Rules								
Conditions	**1**	**2**	**3**	**4**					n
Condition #1	✓			✓	✓				
Condition #2		✓		✓					
Condition #3			✓		✓				
Actions									
Action #1	✓			✓	✓				
Action #2		✓		✓					
Action #3			✓						
Action #4			✓	✓	✓				
Action #5	✓	✓			✓				

Figure 4: Decision table nomenclature

To illustrate the use of a decision table, consider the following excerpt from a processing narrative for a public utility billing system:

If the customer account is billed using a fixed rate method, a minimum monthly charge is assessed for consumption of less than 100 KWH (kilowatt-hours). Otherwise, computer billing applies a Schedule A rate structure. However, if the account is billed using a variable rate method, a Schedule A rate structure will apply to consumption below 100 KWH, with additional consumption billed according to Schedule B.

Figure 5 illustrates a decision table representation of the preceding narrative. Each of the five rules indicates one of five viable conditions (i.e., a T (true) in both fixed rate and variable rate account makes no sense in the context of this procedure; therefore, this condition is omitted). As a general rule, the decision table can be effectively used to supplement other procedural design notation.

	Rules				
Conditions	**1**	**2**	**3**	**4**	**5**
Fixed rate acct.	T	T	F	F	F
Variable rate acct.	F	F	T	T	F
Consumption <100 kwh	T	F	T	F	
Consumption ≥100 kwh	F	T	F	T	
Actions					
Min. monthly charge	✓				
Schedule A billing		✓	✓		
Schedule B billing				✓	
Other treatment					✓

Figure **5:** Resultant decision table

11.2.3 Program Design Language

Program design language (PDL), also called *structured English* or *pseudocode,* is "a pidgin language in that it uses the vocabulary of one language (i.e., English) and the overall syntax of another (i.e., a structured programming language)".

At first glance PDL looks like a modern programming language. The difference between PDL and a real programming language lies in the use of narrative text (e.g., English) embedded directly within PDL statements. Given the use of narrative text embedded directly into a syntactical structure, PDL cannot be compiled (at least not yet).

A program design language may be a simple transposition of a language such as Ada or C. Alternatively, it may be a product purchased specifically for procedural design. Regardless of origin, a design language should have the following characteristics:

- A fixed syntax of keywords that provide for all structured constructs, data declaration, and modularity characteristics.
- A free syntax of natural language that describes processing features.

- Data declaration facilities that should include both simple (scalar, array) and complex (linked list or tree) data structures.
- Subprogram definition and calling techniques that support various modes of interface description.

A basic PDL syntax should include constructs for subprogram definition, interface description, data declaration, techniques for block structuring, condition constructs, repetition constructs, and I/O constructs. The format and semantics for some of these PDL constructs are presented in the section that follows.

11.2.4 A PDL Example

To illustrate the use of PDL, an example of a procedural design is presented for the *SafeHome* security system software introduced in earlier lectures. The system monitors alarms for fire, smoke, burglar, water, and temperature (e.g., furnace breaks while homeowner is away during winter) and produces an alarm bell and calls a monitoring service, generating a voice-synthesized message. In the PDL that follows, we illustrate some of the important constructs noted in earlier sections.
Recall that PDL is not a programming language. The designer can adapt as required without worry of syntax errors. The following PDL defines an elaboration of the procedural design for the *security monitor* component.

```
PROCEDURE security.monitor;
INTERFACE RETURNS system.status;
TYPE signal IS STRUCTURE DEFINED
        name IS STRING LENGTH VAR;
        address IS HEX device location;
        bound.value IS upper bound SCALAR;
        message IS STRING LENGTH VAR;
END signal TYPE;
TYPE system.status IS BIT (4);
TYPE alarm.type DEFINED
        smoke.alarm IS INSTANCE OF signal;
        fire.alarm IS INSTANCE OF signal;
        water.alarm IS INSTANCE OF signal;
```

```
     temp.alarm IS INSTANCE OF signal;
     burglar.alarm IS INSTANCE OF signal;
TYPE phone.number IS area code + 7-digit number;
     •
     •
     •
initialize all system ports and reset all hardware;
CASE OF control.panel.switches (cps):
     WHEN cps = "test" SELECT
          CALL alarm PROCEDURE WITH "on" for test.time in seconds;
     WHEN cps = "alarm-off" SELECT
          CALL alarm PROCEDURE WITH "off";
     WHEN cps = "new.bound.temp" SELECT
     CALL keypad.input PROCEDURE;
     WHEN cps = "burglar.alarm.off" SELECT deactivate signal [burglar.alarm];
     •
     •
     •
     DEFAULT none;
ENDCASE
REPEAT UNTIL activate.switch is turned off
     reset all signal.values and switches;
     DO FOR alarm.type = smoke, fire, water, temp, burglar;
          READ address [alarm.type] signal.value;
          IF signal.value > bound [alarm.type]
          THEN phone.message = message [alarm.type];
               set alarm.bell to "on" for alarm.timeseconds;
               PARBEGIN
               CALL alarm PROCEDURE WITH "on", alarm.time in seconds;
               CALL phone PROCEDURE WITH message [alarm.type],
               phone.number;
               ENDPAR
          ELSE skip
          ENDIF
     ENDFOR
ENDREP
END security.monitor
```

Note that the designer for the *security.monitor* component has used a new construct PARBEGIN . . . ENDPAR that specifies a parallel block. All tasks specified within the PARBEGIN block are executed in parallel. In this case, implementation details are not considered.