COMPONENT-LEVEL DESIGN

Component-level design, also called procedural design, occurs after data, architectural, and interface designs have been established. The intent is to translate the design model into operational software.

What is it?

A complete set of software components is defined during architectural design. But the internal data structures and processing details of each component are not represented at a level of abstraction that is close to the code. Component-level design defines the data structures, algorithms, interface characteristics, and communication mechanisms allocated to each software component.

Who does it?

A software engineer performs component-level design.

Why is it important?

You have to be able to determine whether the program will work before you build it. It is possible to represent the component-level design using a programming language. An alternative approach is to represent the procedural design using some intermediate (e.g., graphical, tabular, or text-based) representation that can be translated easily into source code. The component-level design will reduce the number of errors introduced during coding.

A traditional component, also called a *module*, resides within the software architecture and serves one of three important roles:

(1) a control component that coordinates the invocation of all other problem domain components,

(2) a problem domain component that implements a complete or partial function that is required by the customer,

(3) an infrastructure component that is responsible for functions that support the processing required in the problem domain.

Traditional software components are derived from the analysis model. In this case, the data flow-oriented element of the analysis model serves as the basis for the derivation. Each transform (bubble) represented at the lowest levels of the data flow diagram is mapped into a module hierarchy (lecture 5). Control components (modules) reside near the top of the hierarchy (program architecture), and problem domain components tend to reside toward the bottom of the hierarchy.

STRUCTURED PROGRAMMING

A set of constrained logical constructs were proposed to be used to form any program. 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. The 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 structured constructs reduces program complexity and thereby enhances readability, testability, and maintainability.

Graphical Design Notation

There is no question that graphical tools, such as the flowchart or box diagram, provide useful pictorial patterns that readily depict procedural detail. However, if graphical tools are misused, the wrong picture may lead to the wrong software.

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 structured constructs.

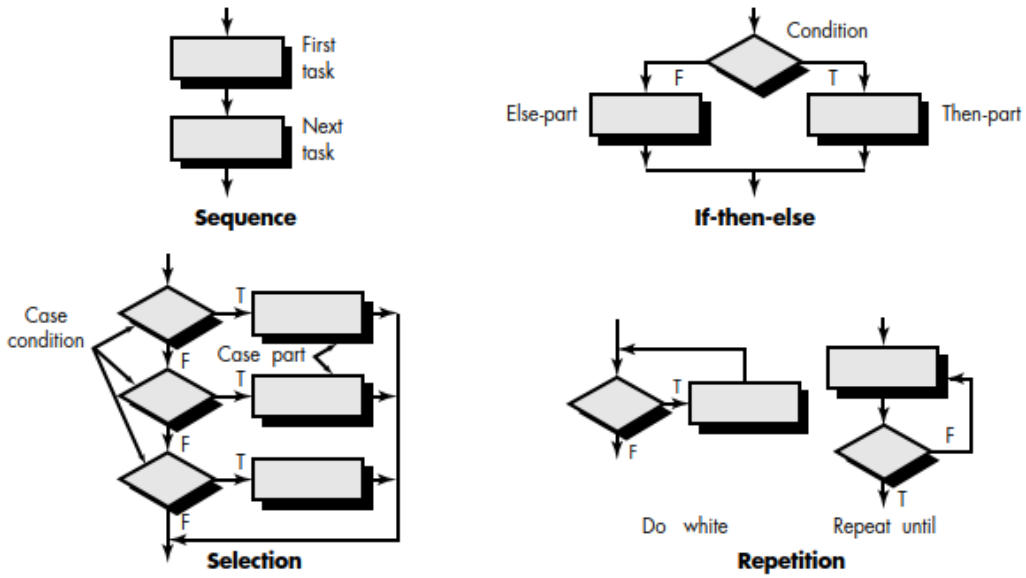


Figure 7.1: Flowchart constructs

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 figure 1 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.

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. The diagrams (also called Nassi-Shneiderman charts, N-S charts, or Chapin charts) have the following characteristics:

(1) functional domain (that is, the scope of repetition or if-then-else) is well defined and clearly visible as a pictorial representation

(2) arbitrary transfer of control is impossible

(3) the scope of local and/or global data can be easily determined

(4) recursion is easy to represent.

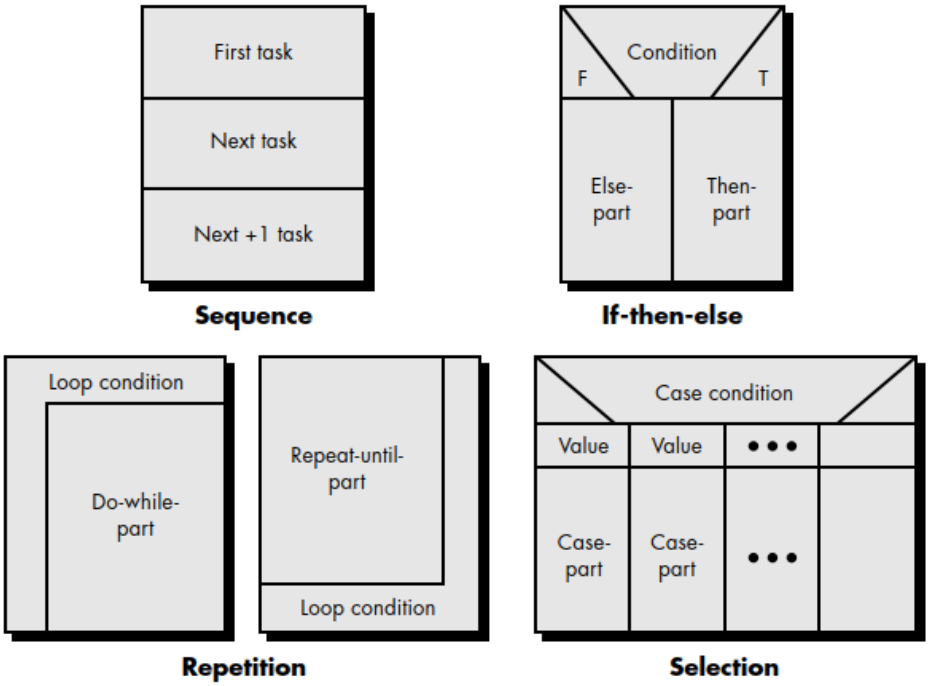
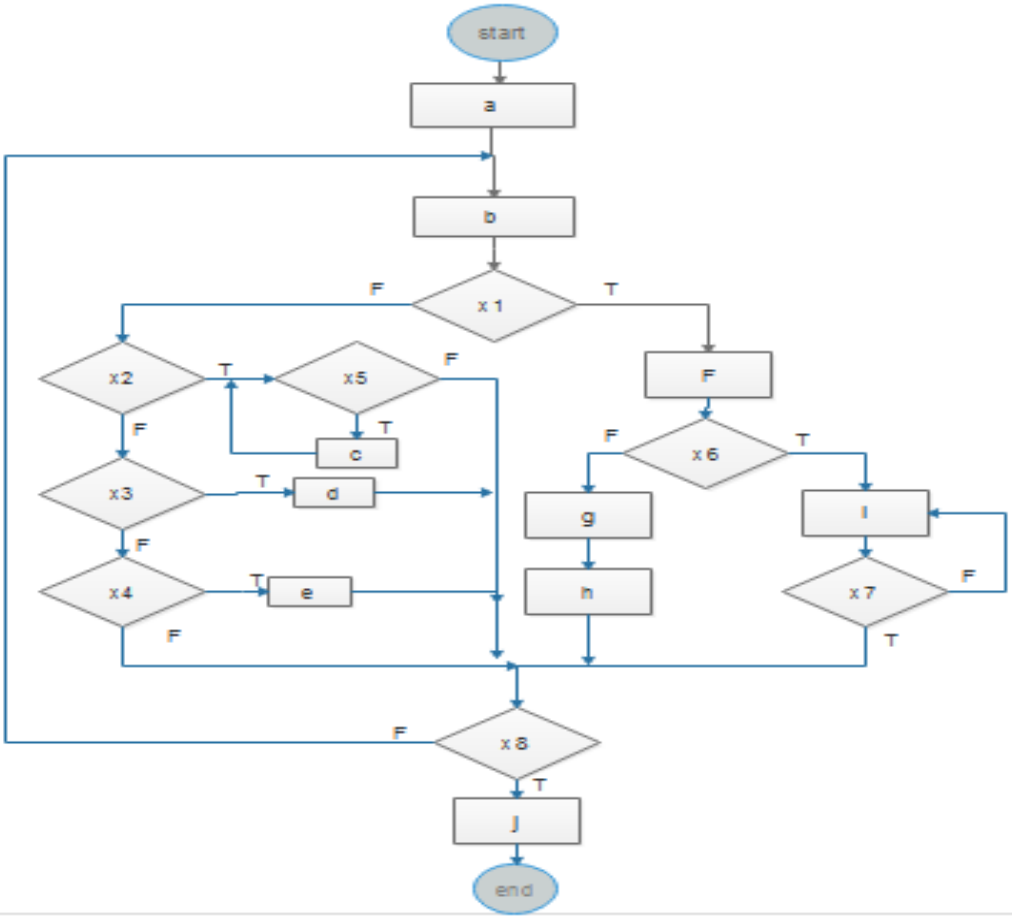
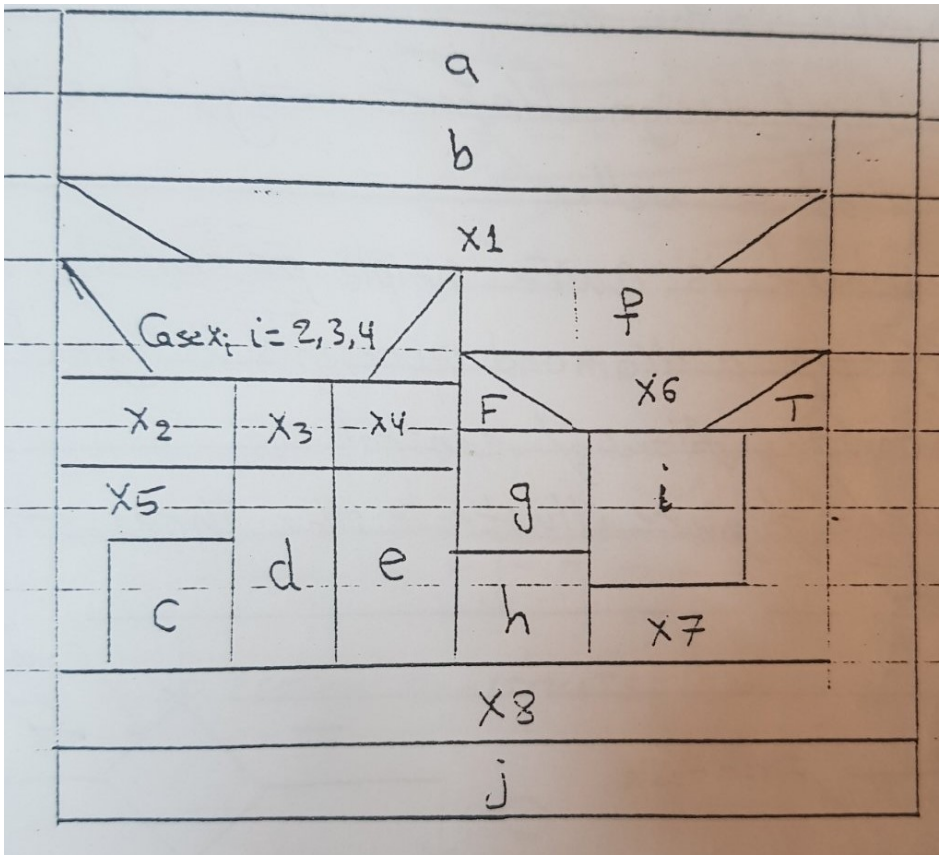


Figure 7.2: Box diagram constructs

Example:





Program design language (PDL)

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)". PDL is used as a generic reference for a design language. At first sight, 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).

```
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;
        •
        •
        •
```

The following attributes of design notation have been established in the context of the general characteristics described previously:

1. **Modularity**: Design notation should support the development of modular software and provide a means for interface specification.
2. **Overall simplicity**: Design notation should be relatively simple to learn, relatively easy to use, and generally easy to read.
3. **Ease of editing**: The procedural design may require modification as the software process proceeds. The ease with which a design representation can be edited can help facilitate each software engineering task.
4. **Data representation:** The ability to represent local and global data is an essential element of component-level design. Ideally, design notation should represent such data directly.
5. **"Code-to" ability**: The software engineering task that follows component-level design is code generation. Notation that may be converted easily to source code reduces effort and error.