Design Engineering

Chapter 9

Pressman

Purpose of Design

- Design is where customer requirements, business needs, and technical considerations all come together in the formulation of a product or system.
- The design model provides detail about the software data structures, architecture, interfaces, and components.
- The design model can be assessed for quality and be improved before code is generated and tests are conducted.

Purpose of Design

- Software design is an <u>iterative process</u> through which requirements are translated into a blueprint for constructing the software.

- Design begins at a <u>high level</u> of abstraction that can be directly traced back to the <u>data, functional, and behavioral</u> requirements

Four Design Models

- The data/class design transforms analysis classes into design classes along with the data structures required to implement the software.
- The architectural design defines the relationship between major structural elements of the software; architectural styles and design patterns help achieve the requirements defined for the system.

Four Design Models

- The interface design describes how the software communicates with systems that interoperate with it and with humans that use it.
- The component-level design transforms structural elements of the software architecture into a procedural description of software components

From Analysis Model to Design Model

Component-level Design

(Class-based model, Flow-oriented model
Behavioral model)

Interface Design

(Scenario-based model, Flow-oriented model
Behavioral model)

Architectural Design

(Class-based model, Flow-oriented model)

Data/Class Design

(Class-based model, Behavioral model)

Task Set for Software Design

1) Examine the information domain model and design appropriate data structures for data objects and their attributes
2) Using the analysis model, select an architectural style (and design patterns) that are appropriate for the software
3) Partition the analysis model into design subsystems and allocate these subsystems within the architecture
    a) Design the subsystem interfaces
    b) Allocate analysis classes or functions to each subsystem
4) Create a set of design classes or components
    a) Translate each analysis class description into a design class
    b) Check each design class against design criteria; consider inheritance issues
    c) Define methods associated with each design class
    d) Evaluate and select design patterns for a design class or subsystem

Task Set for Software Design

5) Design any interface required with external systems or devices
6) Design the user interface
7) Conduct component-level design
    a) Specify all algorithms at a relatively low level of abstraction
    b) Refine the interface of each component
    c) Define component-level data structures
    d) Review each component and correct all errors uncovered
8) Develop a deployment model
    - Show a physical layout of the system, revealing which components will be located where in the physical computing environment

Design Quality

Quality's Role

- The importance of design is quality
- Design is the place where quality is fostered
  - Provides representations of software that can be assessed for quality
  - Accurately translates a customer's requirements into a finished software product or system
  - Serves as the foundation for all software engineering activities that follow
- Without design, we risk building an unstable system that
  - Will fail when small changes are made
  - May be difficult to test
  - Cannot be assessed for quality later in the software process when time is short and most of the budget has been spent
- The quality of the design is assessed through a series of formal technical reviews or design walkthroughs

Goals of a Good Design

- The design must <u>implement</u> all of the <u>explicit</u> requirements contained in the analysis model
  - It must also accommodate all of the <u>implicit</u> requirements desired by the customer
- The design must be a <u>readable and understandable guide</u> for those who generate code, and for those who test and support the software
- The design should provide a <u>complete picture</u> of the software, addressing the data, functional, and behavioral domains from an implementation perspective

"Writing a clever piece of code that works is one thing; designing something that can support a long-lasting business is quite another."

Design Quality Guidelines

1) A design should exhibit an <u>architecture</u> that
   a) Has been created using recognizable <u>architectural styles or patterns</u>
   b) Is composed of components that exhibit good design characteristics
   c) Can be implemented in an <u>evolutionary</u> fashion, thereby facilitating implementation and testing
2) A design should be <u>modular</u>; that is, the software should be logically partitioned into elements or subsystems
3) A design should contain <u>distinct representations</u> of data, architecture, interfaces, and components
4) A design should lead to <u>data structures</u> that are <u>appropriate</u> for the classes to be implemented and are drawn from recognizable data patterns

Quality Guidelines

5) A design should lead to <u>components</u> that exhibit <u>independent</u> functional characteristics
6) A design should lead to interfaces that <u>reduce the complexity of connections</u> between components and with the external environment
7) A design should be derived using a repeatable method that is <u>driven by</u> information obtained during software <u>requirements analysis</u>
8) A design should be represented using a <u>notation</u> that effectively communicates its meaning

"Quality isn't something you lay on top of subjects and objects like tinsel on a Christmas tree."

Design Concepts

- **Abstraction**
  - Procedural abstraction – a sequence of instructions that have a specific and limited function
  - Data abstraction – a named collection of data that describes a data object
- **Architecture**
  - The overall structure of the software and the ways in which the structure provides conceptual integrity for a system
  - Consists of components, connectors, and the relationship between them
- **Patterns**
  - A design structure that solves a particular design problem within a specific context
  - It provides a description that enables a designer to determine whether the pattern is applicable, whether the pattern can be reused, and whether the pattern can serve as a guide for developing similar patterns

Design Concepts

- **Modularity**
  - Separately named and addressable components (i.e., modules) that are integrated to satisfy requirements (divide and conquer principle)
  - Makes software intellectually manageable so as to grasp the control paths, span of reference, number of variables, and overall complexity
- **Information hiding**
  - The designing of modules so that the algorithms and local data contained within them are inaccessible to other modules
  - This enforces access constraints to both procedural (i.e., implementation) detail and local data structures
- **Functional independence**
  - Modules that have a "single-minded" function and an aversion to excessive interaction with other modules
  - High cohesion – a module performs only a single task
  - Low coupling – a module has the lowest amount of connection needed with other modules

Design Concepts

- **Stepwise refinement**
  - Development of a program by successively refining levels of procedure detail
  - Complements abstraction, which enables a designer to specify procedure and data and yet suppress low-level details
- **Refactoring**
  - A reorganization technique that simplifies the design (or internal code structure) of a component without changing its function or external behaviour
  - Removes redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failures
- **Design classes**
  - Refines the analysis classes by providing design detail that will enable the classes to be implemented
  - Creates a new set of design classes that implement a software infrastructure to support the business solution

Types of Design Classes

- **User interface classes** – define all abstractions necessary for human-computer interaction
- **Business domain classes** – refined from analysis classes; identify attributes and services (methods) that are required to implement some element of the business domain
- **Process classes** – implement business abstractions required to fully manage the business domain classes
- **Persistent classes** – represent data stores (e.g., a database) that will persist beyond the execution of the software
- **System classes** – implement software management and control functions that enable the system to operate and communicate within its computing environment and the outside world

Characteristics of a Well-Formed Design Class

- **Complete and sufficient**
  - Contains the complete encapsulation of all attributes and methods that exist for the class
  - Contains only those methods that are sufficient to achieve the intent of the class
- **Primitiveness**
  - Each method of a class focuses on accomplishing one service for the class
- **High cohesion**
  - The class has a small, focused set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities
- **Low coupling**
  - Collaboration of the class with other classes is kept to an acceptable minimum
  - Each class should have limited knowledge of other classes in other subsystems

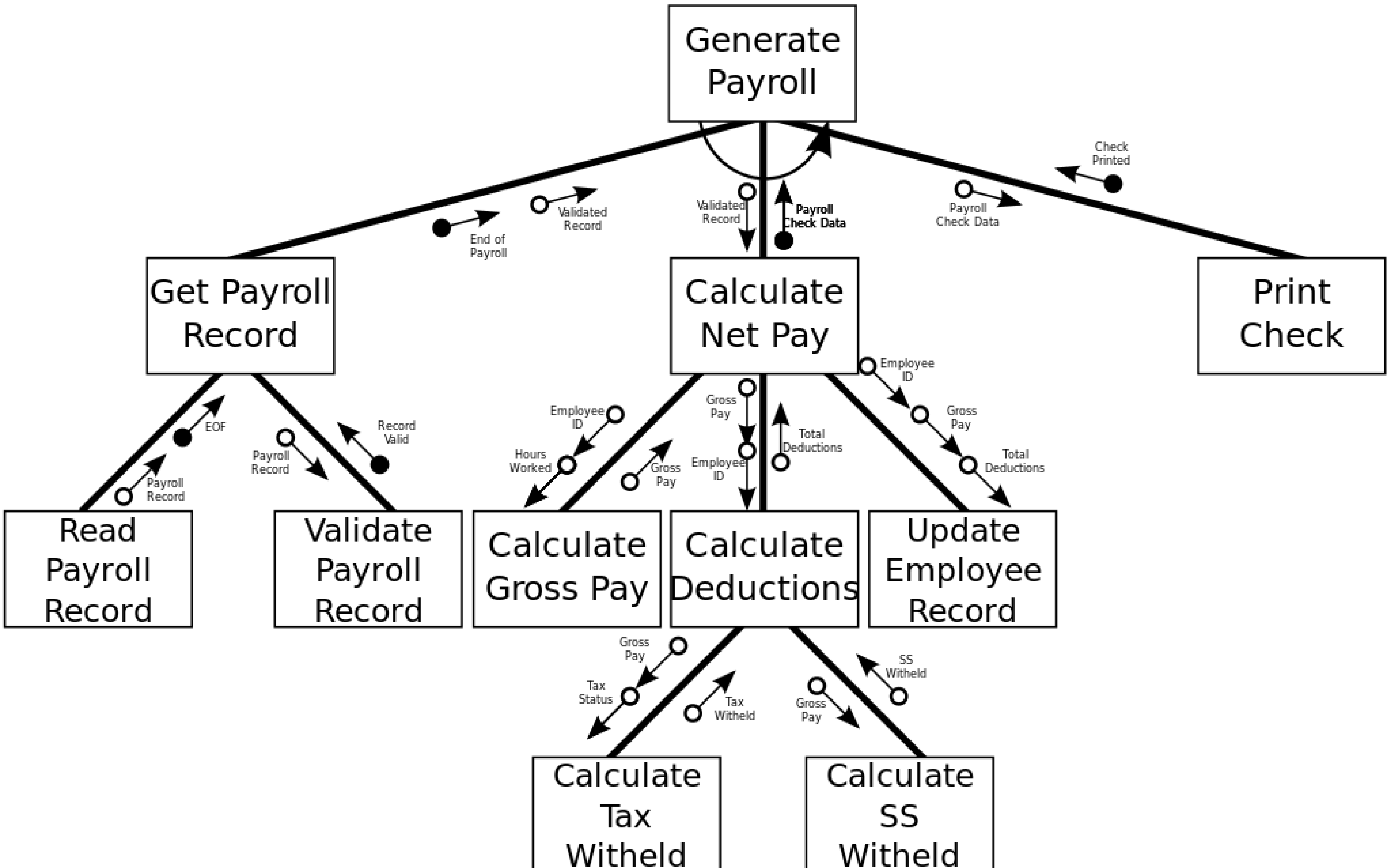
Design Tools

- HIPO Diagram
- Structure Chart
- Decision Tree

Structure Chart

- A **Structure Chart** (SC) in software engineering, is a chart which shows the breakdown of a system to its lowest manageable levels.
- They are used in structured programming to arrange program modules into a tree.
- Each module is represented by a box, which contains the module's name.
- The tree structure visualizes the relationships between modules.

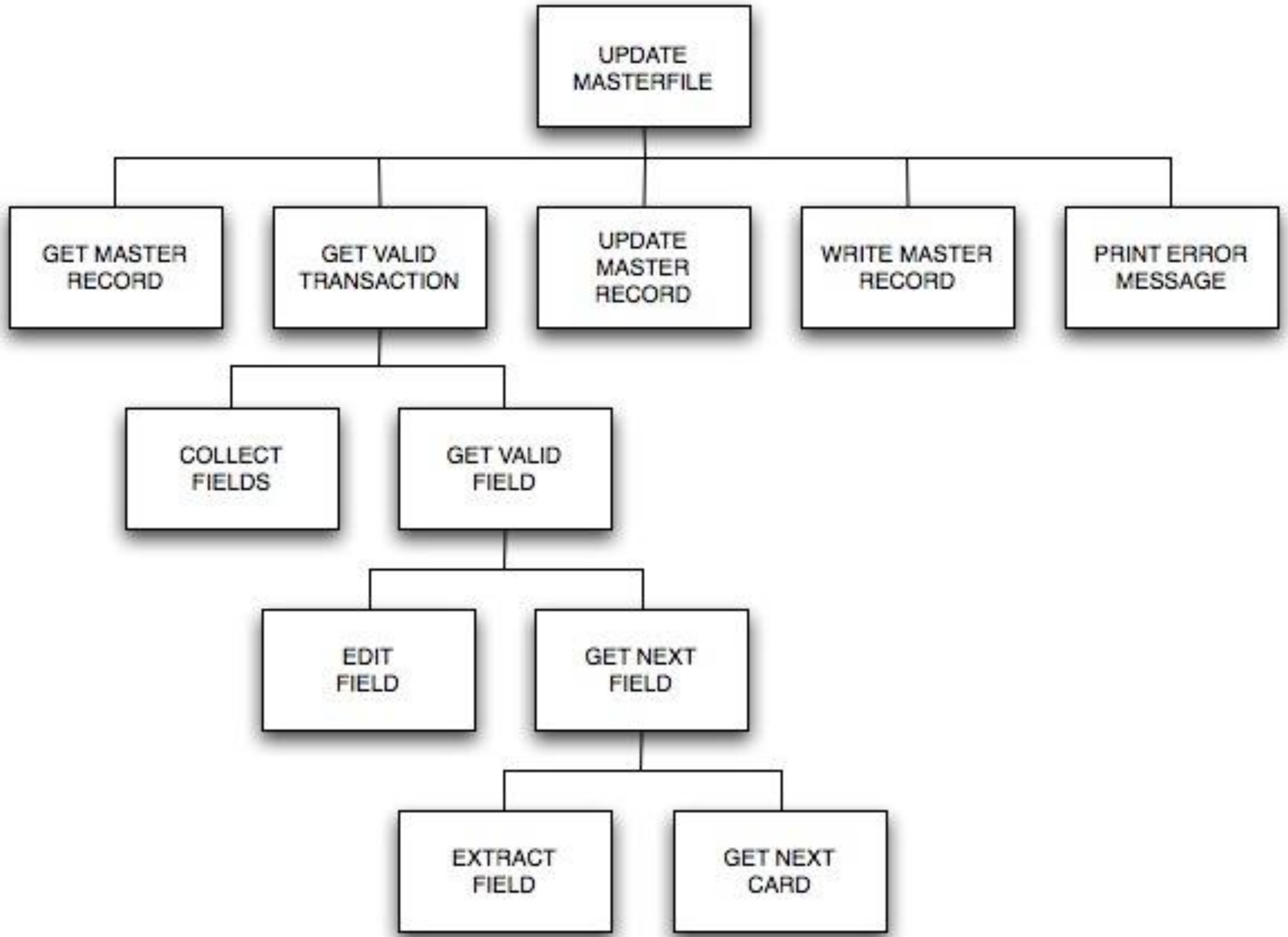
Structure Chart



HIPO Diagram

- **HIPO** model (short for *Hierarchical Input Process Output* model) is a systems analysis design aid and documentation technique from the 1970s, used for representing the modules of a system as a hierarchy and for documenting each module.

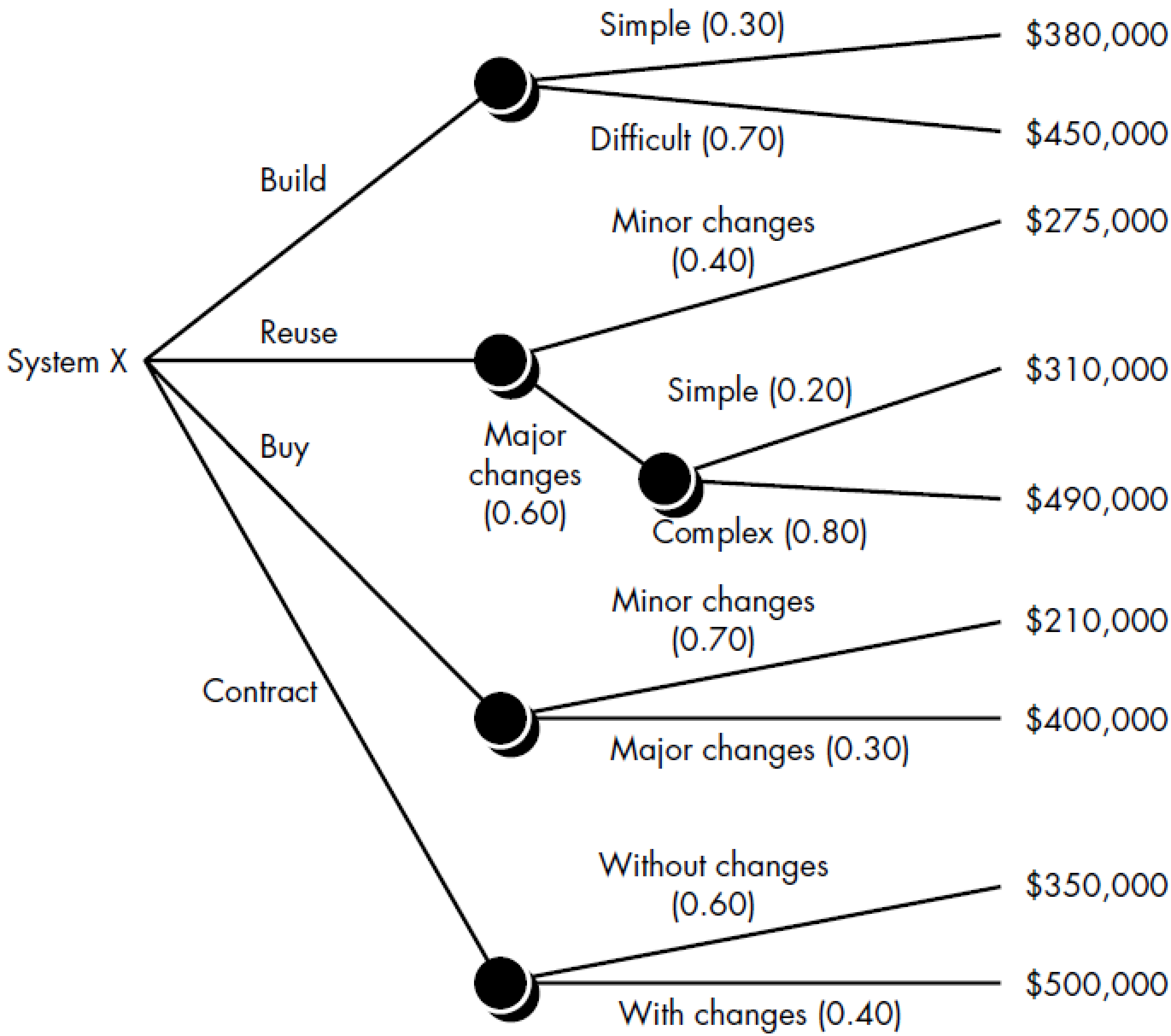
HIPO Diagram



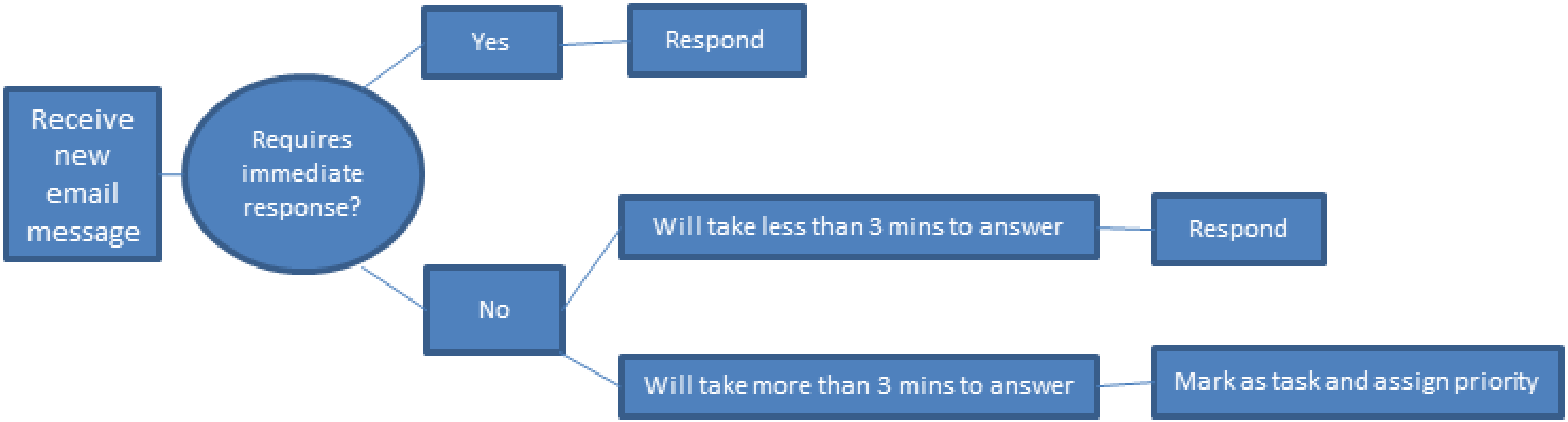
Decision Tree

- A decision tree is a graph that uses a branching method to illustrate every possible outcome of a decision.
- Decision trees can be drawn by hand or created with a graphics program or specialized software.
- Informally, decision trees are useful for focusing discussion when a group must make a decision.
- Programmatically, they can be used to assign monetary/time or other values to possible outcomes so that decisions can be automated.

Decision Tree



Decision Tree



Decision Table

- **Decision tables** are a precise yet compact way to model complex rule sets and their corresponding actions.
- Decision tables, like flowcharts and if-then-else and switch-case statements, associate conditions with actions to perform, but in many cases do so in a more elegant way.

Example

- The limited-entry decision table is the simplest to describe.
- The condition alternatives are simple Boolean values, and the action entries are check-marks, representing which of the actions in a given column are to be performed.
- A technical support company writes a decision table to diagnose printer problems based upon symptoms described to them over the phone from their clients.

Balanced Decision Table

Printer troubleshooter

		Rules							
Conditions	Printer does not print	Y	Y	Y	Y	N	N	N	N
	A red light is flashing	Y	Y	N	N	Y	Y	N	N
	Printer is unrecognised	Y	N	Y	N	Y	N	Y	N
Actions	Check the power cable			X					
	Check the printer-computer cable	X		X					
	Ensure printer software is installed	X		X		X		X	
	Check/replace ink	X	X			X	X		
	Check for paper jam		X		X				

Program embedded decision tables

- Decision tables can be, and often are, embedded within computer programs and used to 'drive' the logic of the program.
- A simple example might be a lookup table containing a range of possible input values and a function pointer to the section of code to process that input.

Static Decision Table

Input	Function Pointer
'1'	Function 1 (initialize)
'2'	Function 2 (process 2)
'9'	Function 9 (terminate)

General Format

1 Policy or Process Name

6 Rules

2 Conditions

3 Condition Alternatives

4 Actions

5 Action Entries

Payroll Policy	**Rules**			
	1	**2**	**3**	**4**
Employee Type	S	H	H	H
Hours Worked	--	<40	=40	>40
Pay Base Salary	X			
Pay Hourly Wage		X	X	X
Pay Overtime				X
Produce Absence Report		X		

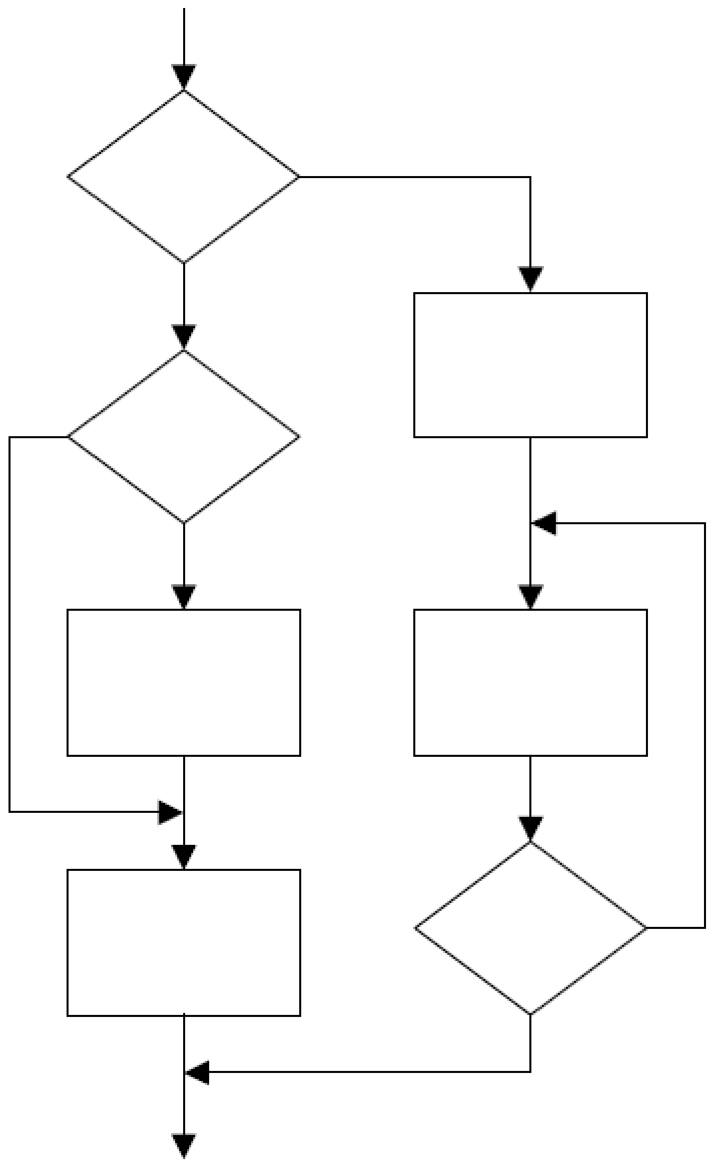
S = Salaried Employee; H = Hourly Employee

Structured Flowcharts

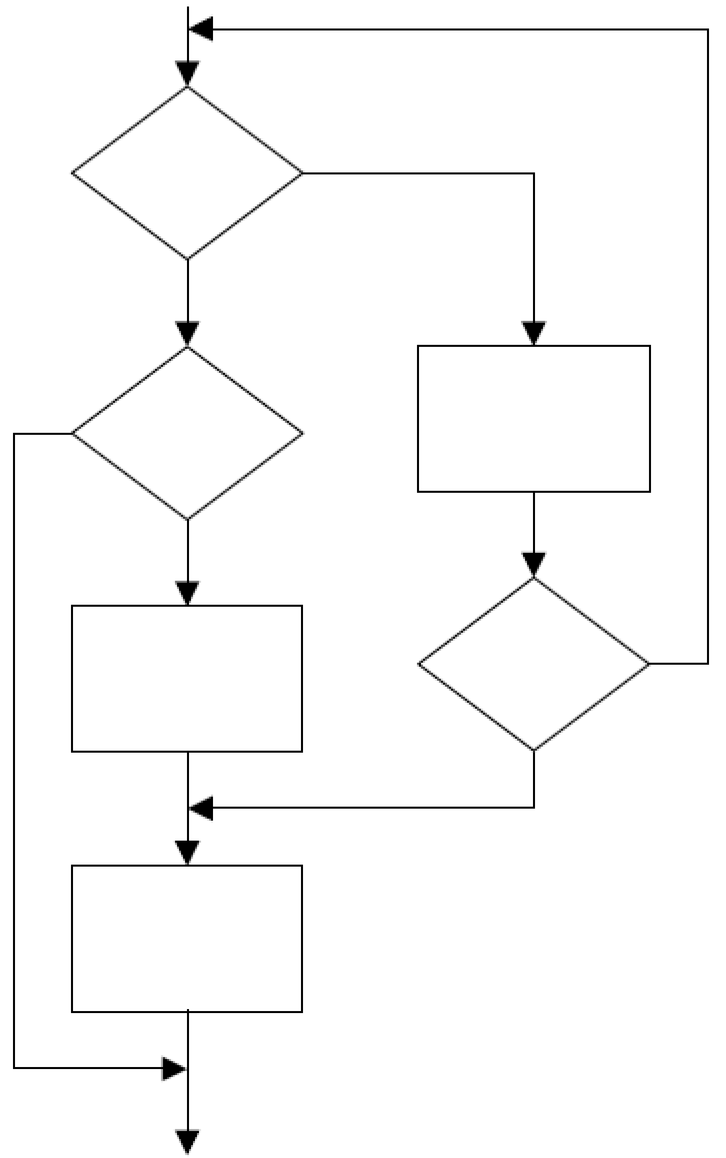
- **Unstructured flow** is often called '**spaghetti ' programming** and normally has elements of its structure impossibly intertwined around other elements.

- A program of this sort is very difficult to understand, implement, debug and maintain.

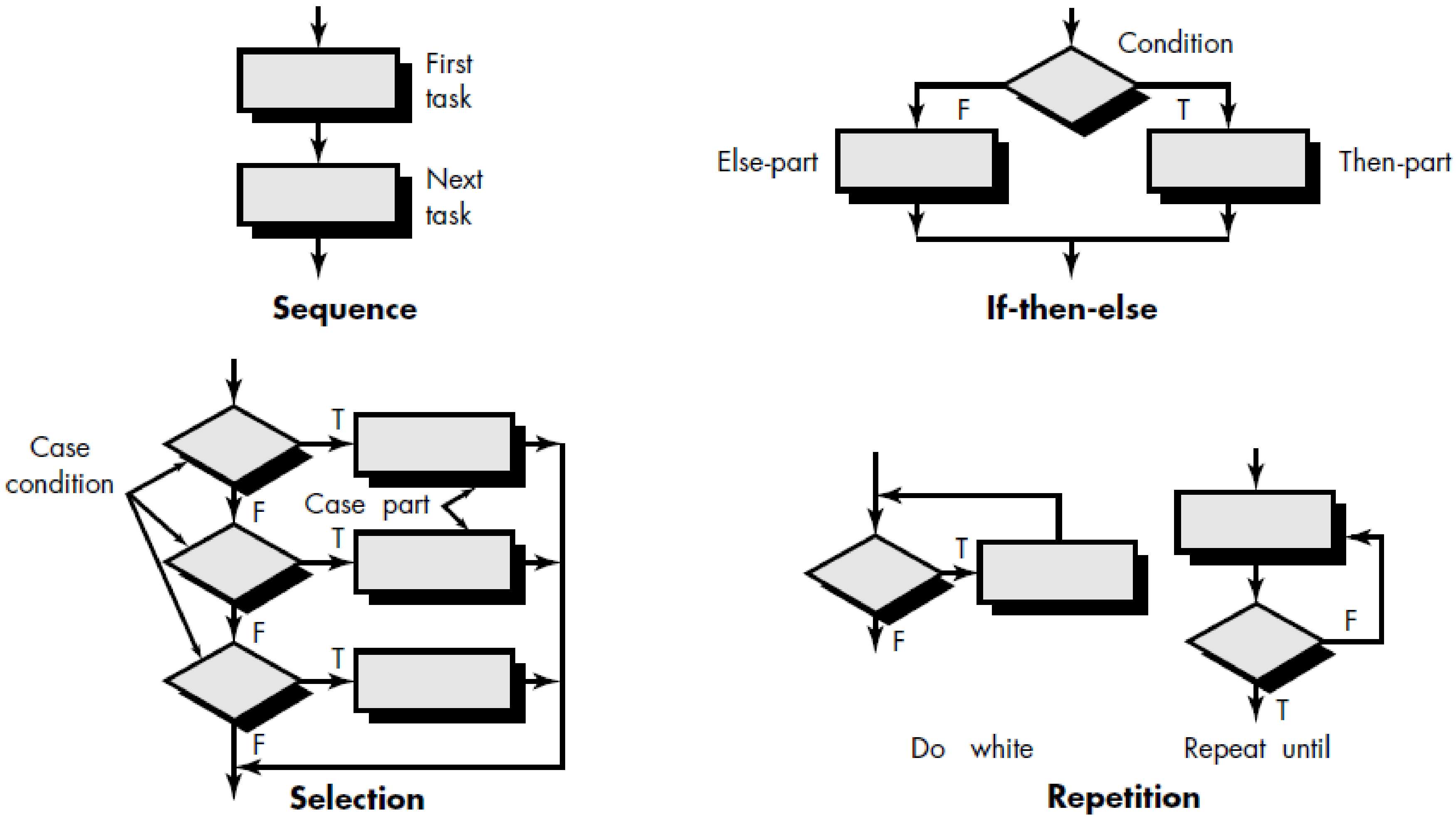
Structured Flowchart



Unstructured Flowchart



Flowchart Constructs



Structured English or pseudo-code or Program design language (PDL)

- PDL is "a program design 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 uses narrative text (e.g., English) embedded directly within PDL statements.
- PDL cannot be compiled.
- However, PDL tools currently exist to translate PDL into a programming language ‘skeleton’ or a graphical representation (flowchart) of design.

PDL features

- A fixed syntax of keywords that provide structured constructs and data declaration.

- 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.

Problem Description

- To illustrate the use of PDL, we present an example of a procedural design for the *SafeHome security system software.*
- *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.

```
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
```

PARBEGIN..ENDPAR

- PARBEGIN . . . ENDPAR specifies a parallel block.
- All tasks specified within the PARBEGIN block are executed in parallel.

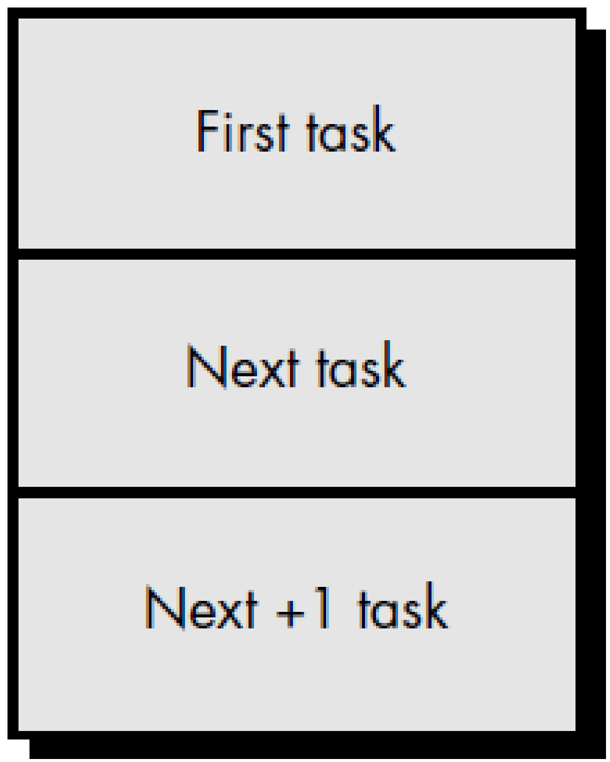
Nassi-Shneiderman Diagram or Box Diagram

- It is evolved from a desire to develop a procedural design representation that would **not allow violation of the structured constructs**.
- It is developed by Nassi and Shneiderman and extended by Chapin, the diagrams are also called *Nassi-Shneiderman charts, N-S charts, or Chapin charts.*

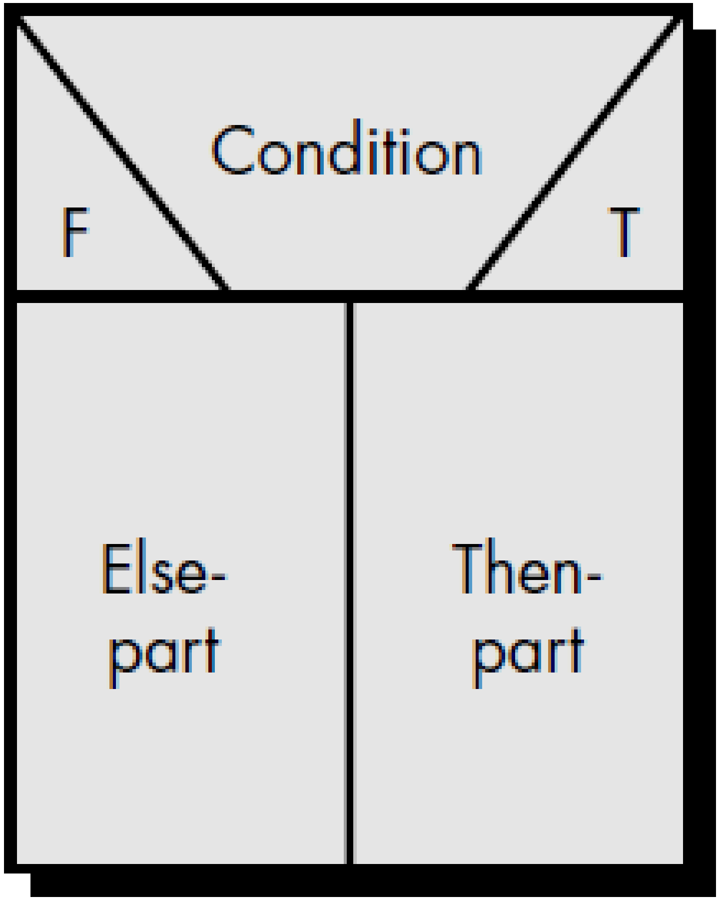
Features

- functional domain (that is, the scope of repetition or if-then-else) is well defined and clearly visible as a pictorial representation.
- arbitrary transfer of control is impossible.
- the scope of local and/or global data can be easily determined.
- recursion is easy to represent.

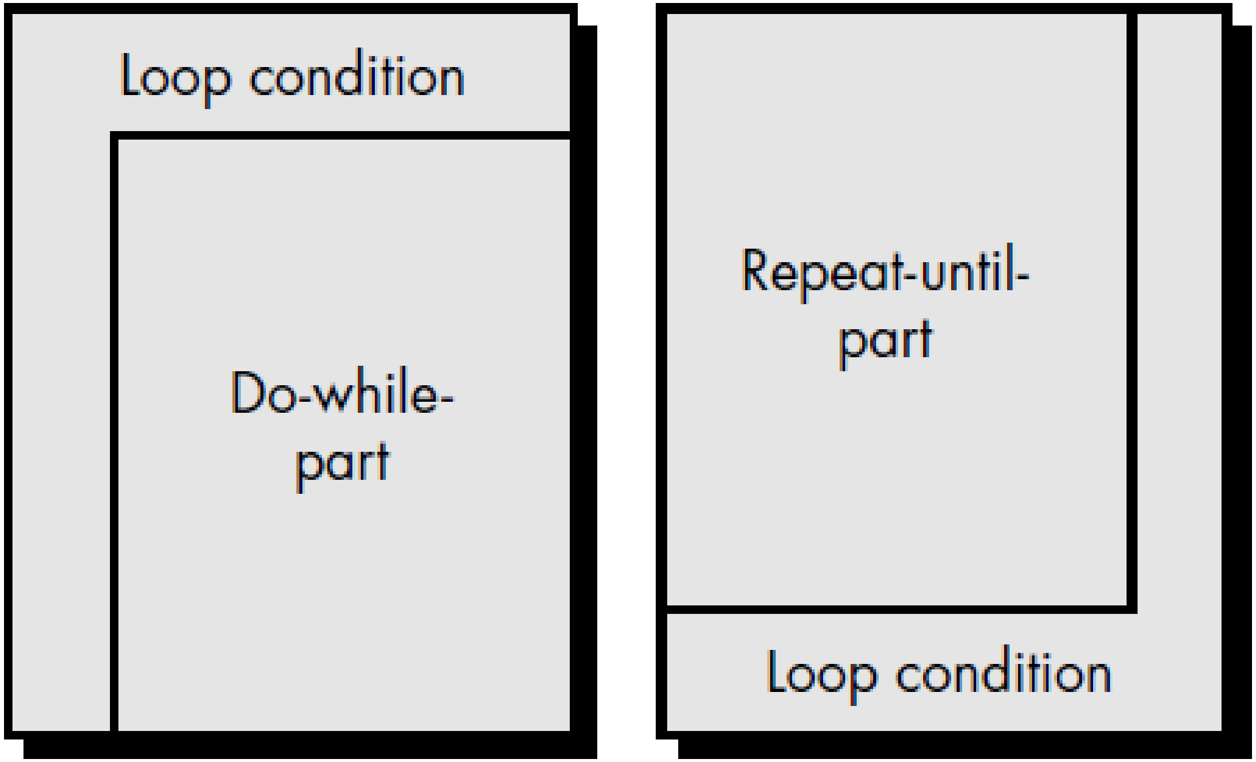
Sequence



If-then-else

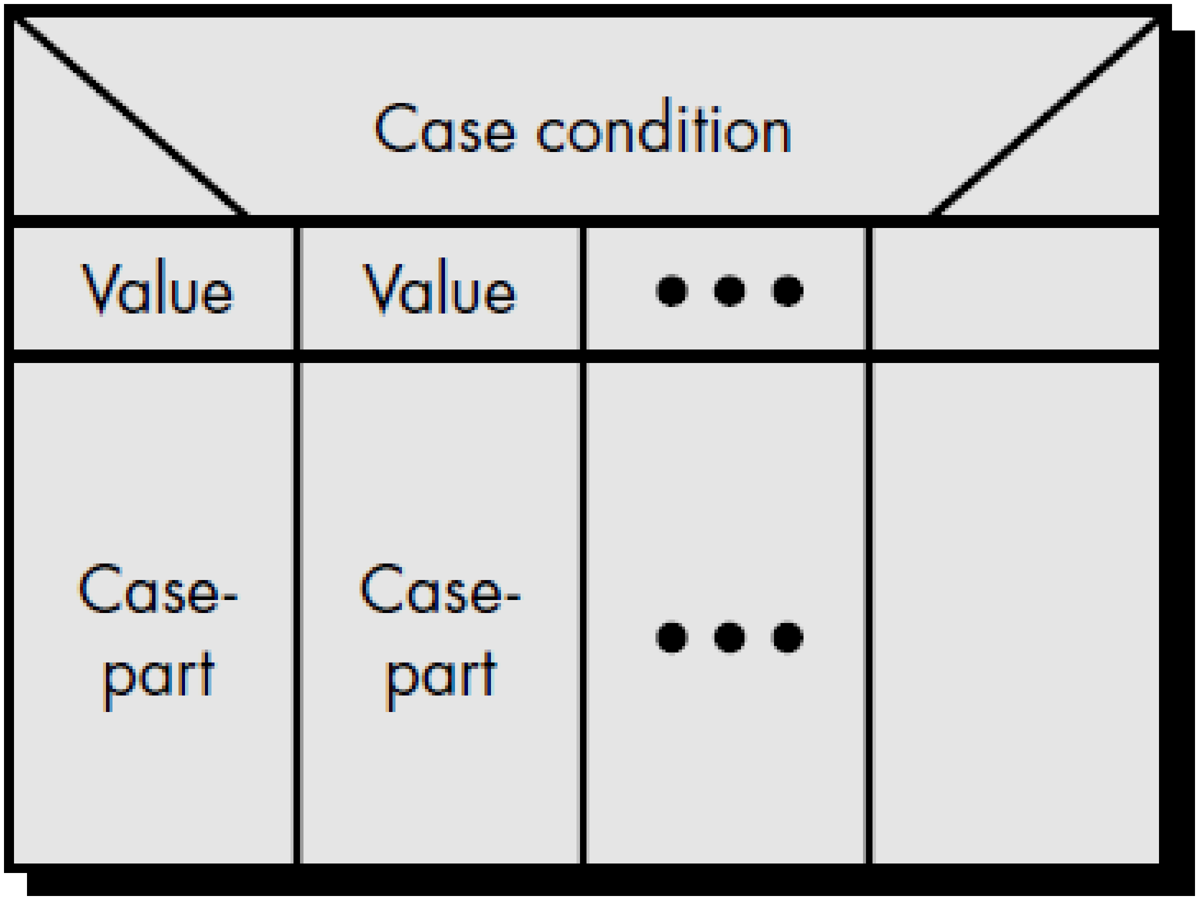


Repetition

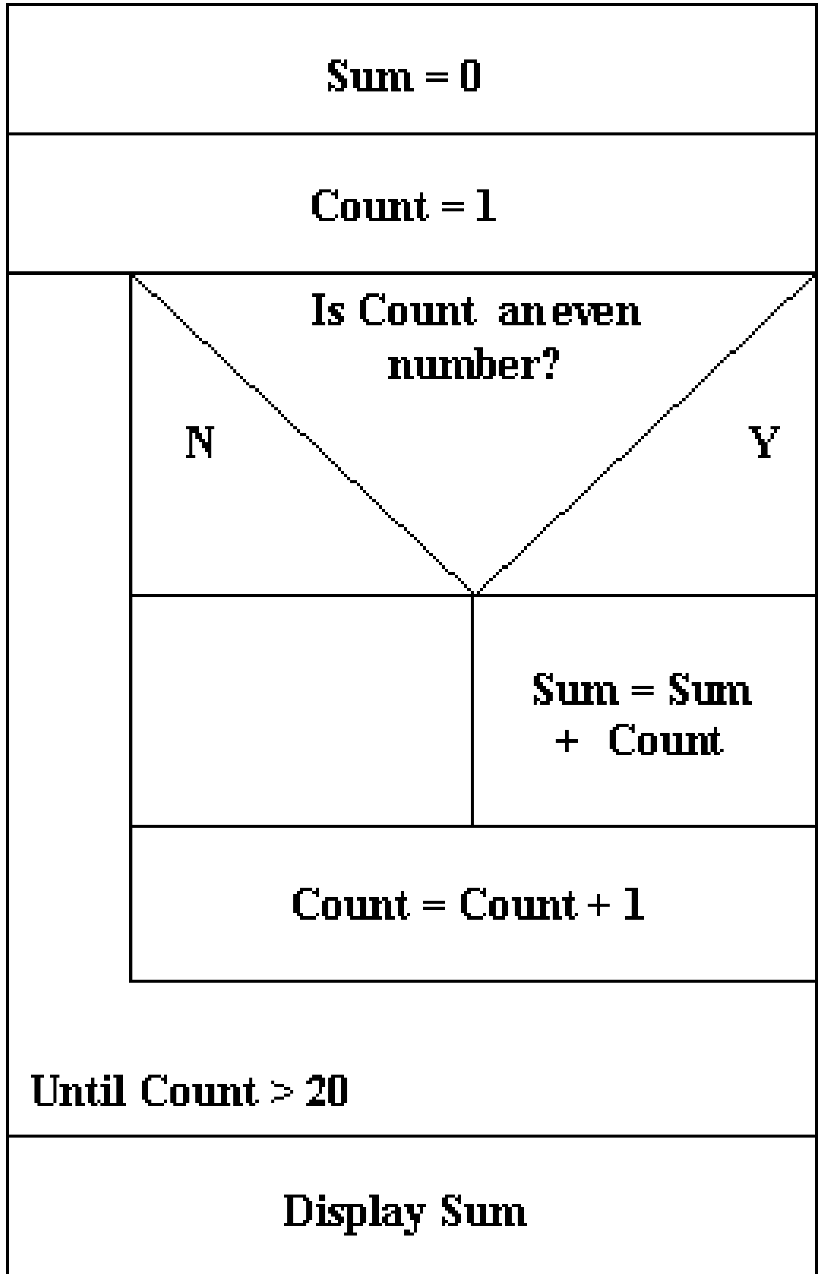


Loop condition
Do-while-
part
Repeat-until-
part
Loop condition

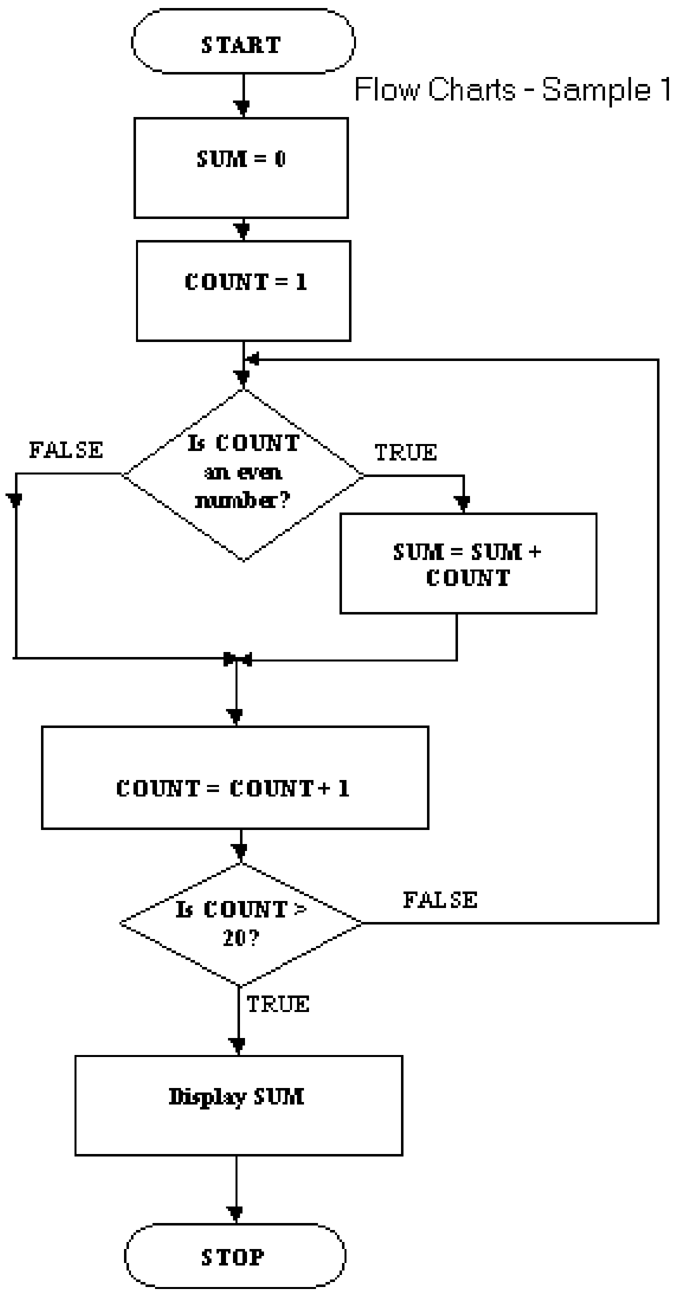
Selection



Box Diagram


Sum = 0
Count = 1
Is Count an even number?
N
Y
Sum = Sum + Count
Count = Count + 1
Until Count > 20
Display Sum

Flowchart


START
Flow Charts - Sample 1
SUM = 0
COUNT = 1
FALSE
Is COUNT an even number?
TRUE
SUM = SUM + COUNT
COUNT = COUNT + 1
Is COUNT > 20?
FALSE
TRUE
Display SUM
STOP

Exercise 1

```
IF x = = 100
 THEN IF x == 120
   THEN IF x == 140
              THEN DISPLAY 'A'
              ELSE DISPLAY 'B'
   ELSE DISPLAY 'C'
 ELSE DISPLAY 'D'
ENDIF
```

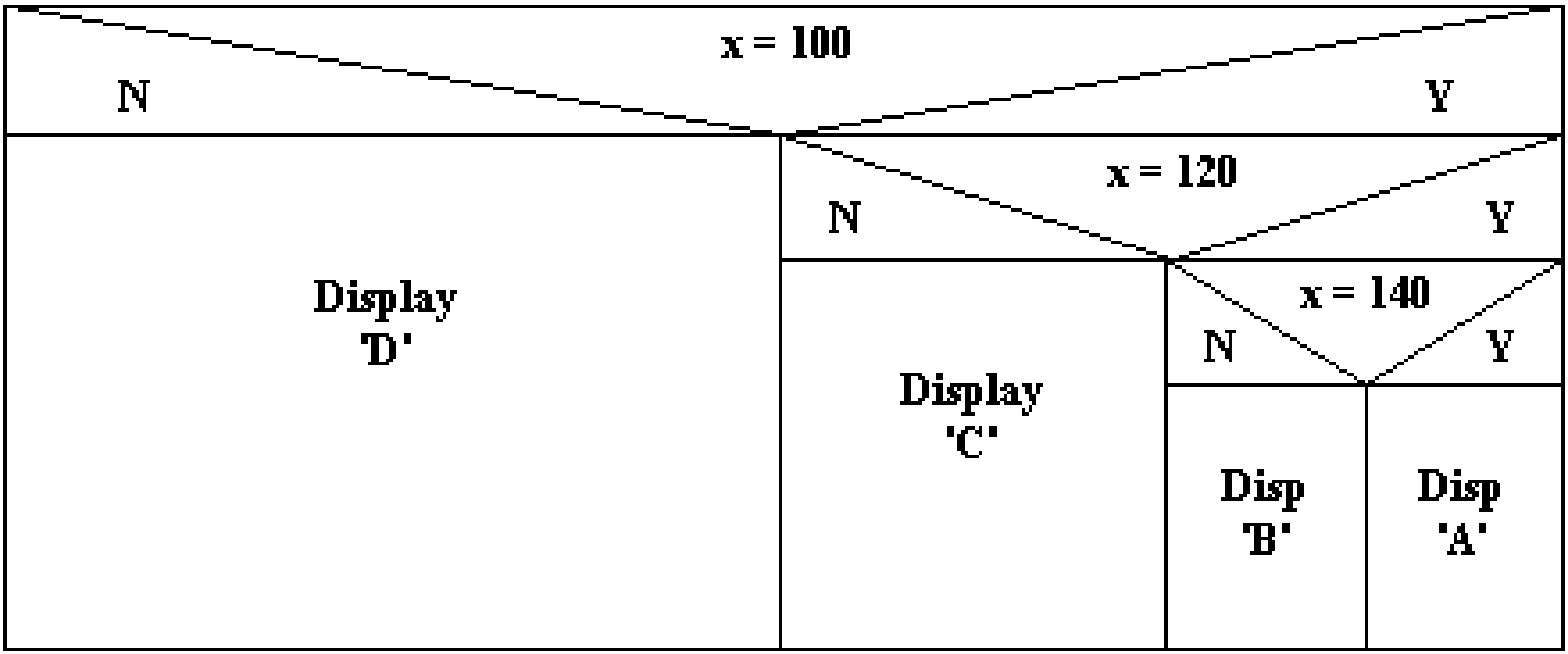
Exercise 2

```
X = 1
REPEAT
 X = X + 1
 DISPLAY X
UNTIL X > 10
```

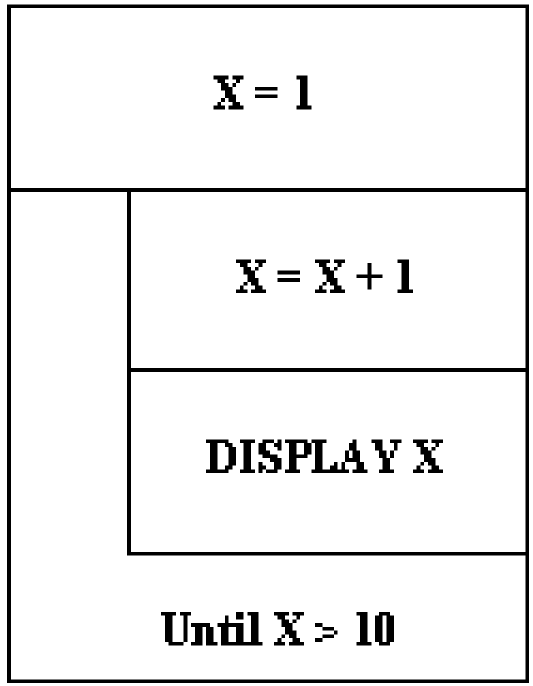
Exercise 3

```
X = 0
Y = 0
DOWHILE X < 10
 DISPLAY X
 REPEAT
  Y = Y + 1
 UNTIL Y > 5
 X = X + 1
 Y = 0
END WHILE
```

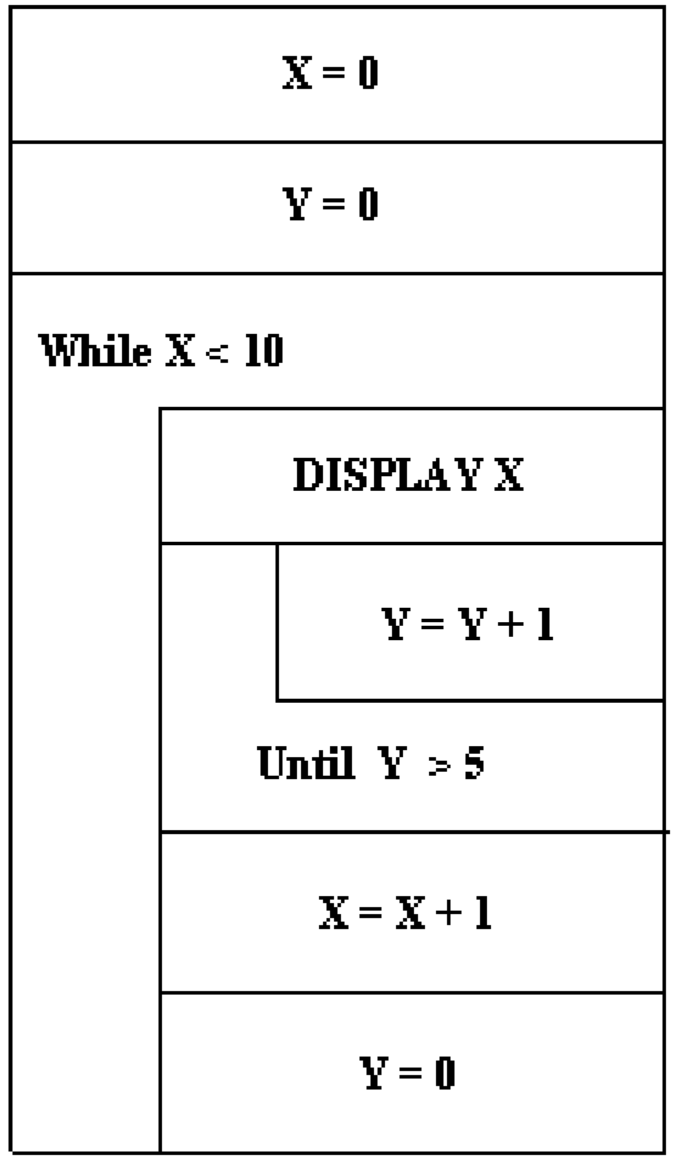
Solution 1



Solution 2



Solution 3



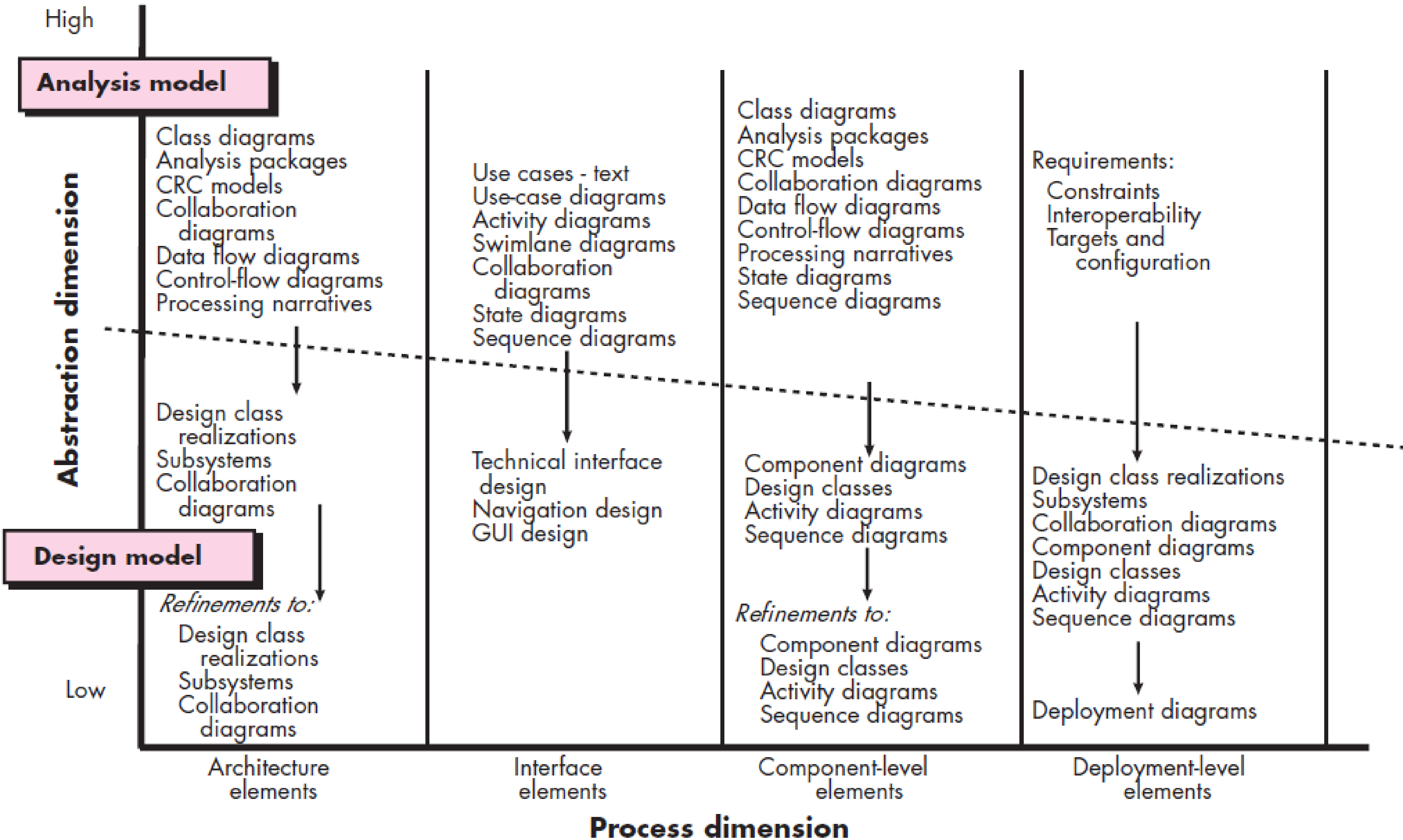
Homework

- Types of coupling
- Types of cohesion

Design Model

- The design model can be viewed in two different dimensions, process dimension and abstraction dimension.
- The ***process dimension*** *indicates the evolution of the design model as design* tasks are executed as part of the software process.
- The ***abstraction dimension*** represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively.

Dimensions of the design model



Four elements of design model

- Data
- Architecture
- Components
- Interface

Difference between analysis model and design model

- The design model use many of the same UML diagrams that were used in the analysis model.
  - Diagrams are refined and elaborated.
  - More implementation-specific detail is provided.
  - Architectural structure and style are highlighted.
  - Components that reside within the architecture are outlined.
  - Interfaces between the components and with the outside world are emphasized.

Design model

- Design model elements are not always developed in a sequential fashion.
- In most cases preliminary architectural design sets the stage and is followed by interface design and component-level design, which often occur in parallel.
- The deployment model is usually delayed until the design has been fully developed.
- Design patterns can be applied at any point during design.

Data Design Elements

- Data design is also known as **data architecting**.
- The structure of data has always been an important part of software design.
- At the **program component level**, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high-quality applications.
- At the **application level**, the translation of a data model into a database is pivotal to achieving the business objectives of a system.
- At the **business level**, the collection of information stored in disparate databases and reorganized into a “data warehouse” enables data mining or knowledge discovery that can have an impact on the success of the business itself.
- In every case, data design plays an important role.

Architectural Design Elements

- The architectural design for software is the equivalent to the **floor plan of a house**.
- The architectural model is derived from three sources:
  - Information about the application domain for the software to be built.
  - Specific requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand.
  - The availability of architectural styles and patterns .

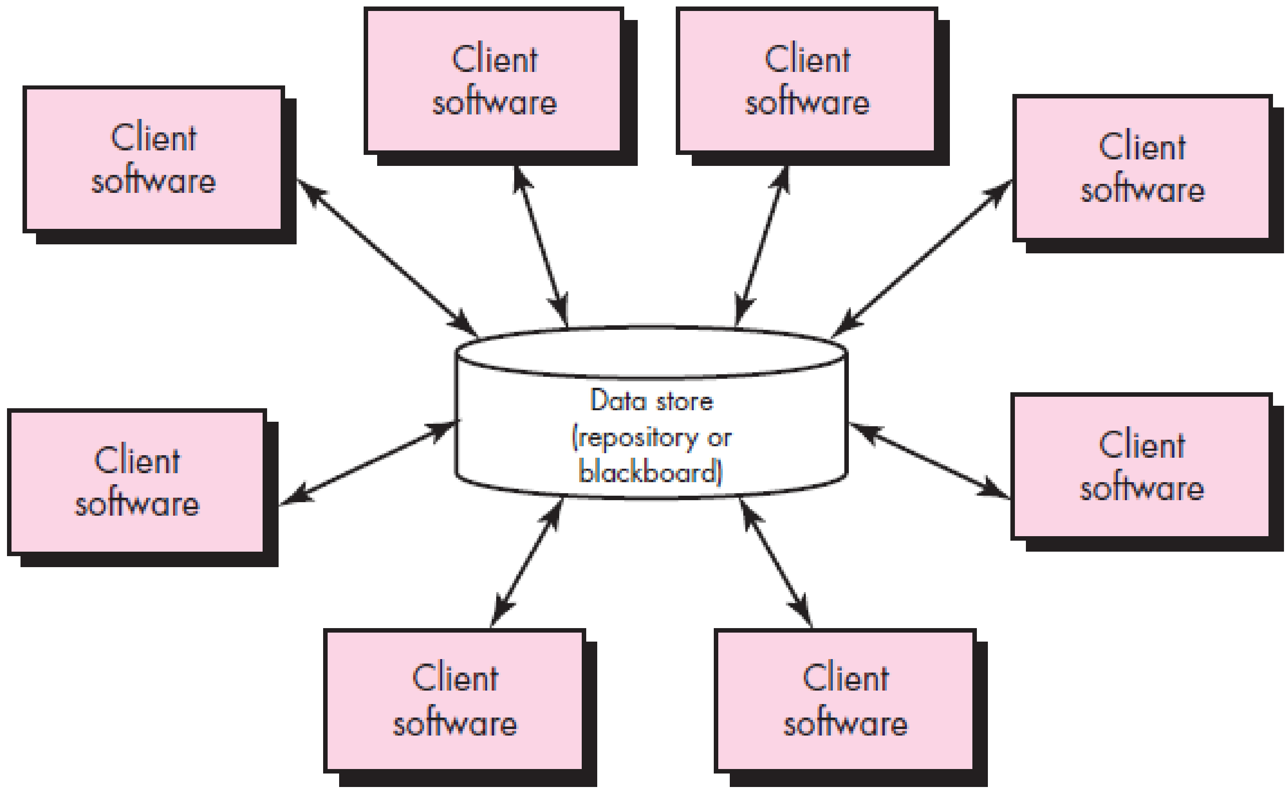
Architectural genres

- AI
- Commercial and non-profit
- Communications
- Content Authoring
- Devices
- Entertainment and Sports
- Financial
- Games
- Government
- Industrial
- Legal
- Medical
- Military
- Operating Systems
- Platforms
- Scientific
- Tools
- Transportation
- Utilities

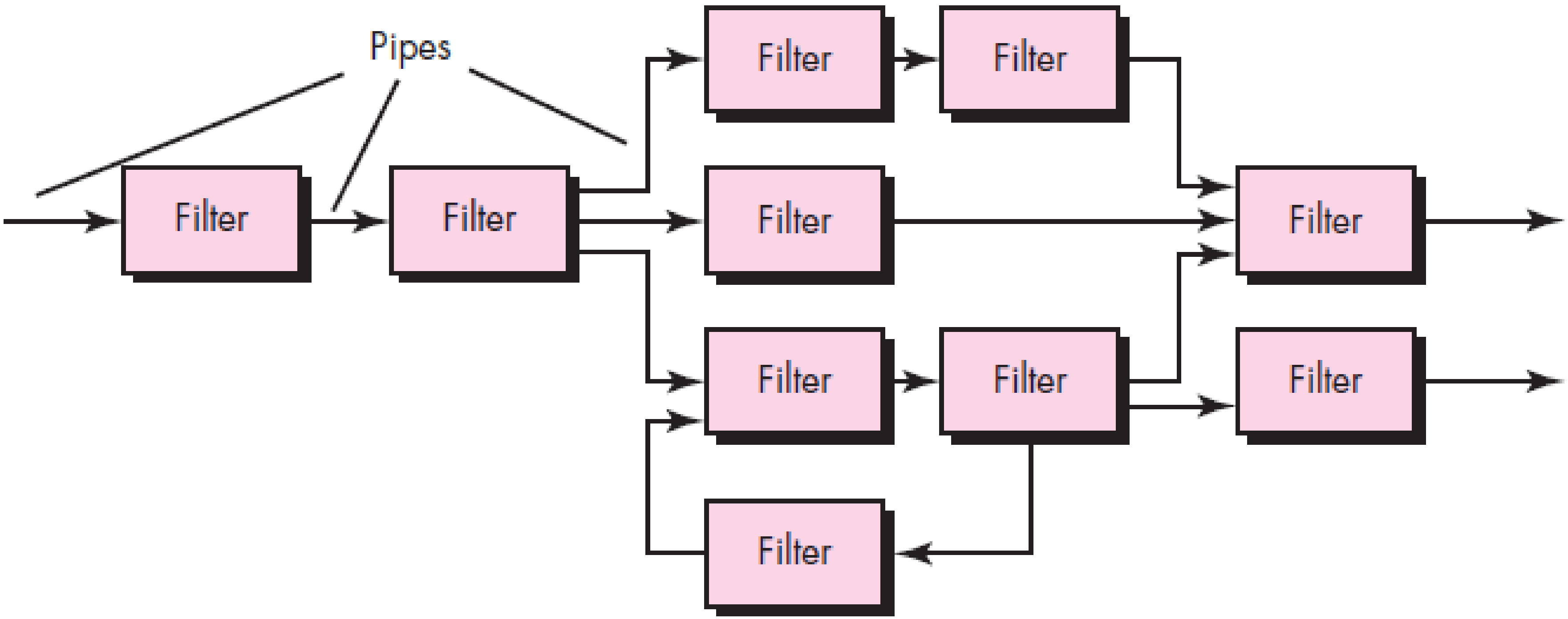
Architectural Styles

- Data-centered architectures
- Data-flow architectures
- Call and return architectures
- Object-oriented architectures
- Layered architectures

Data-centered architecture

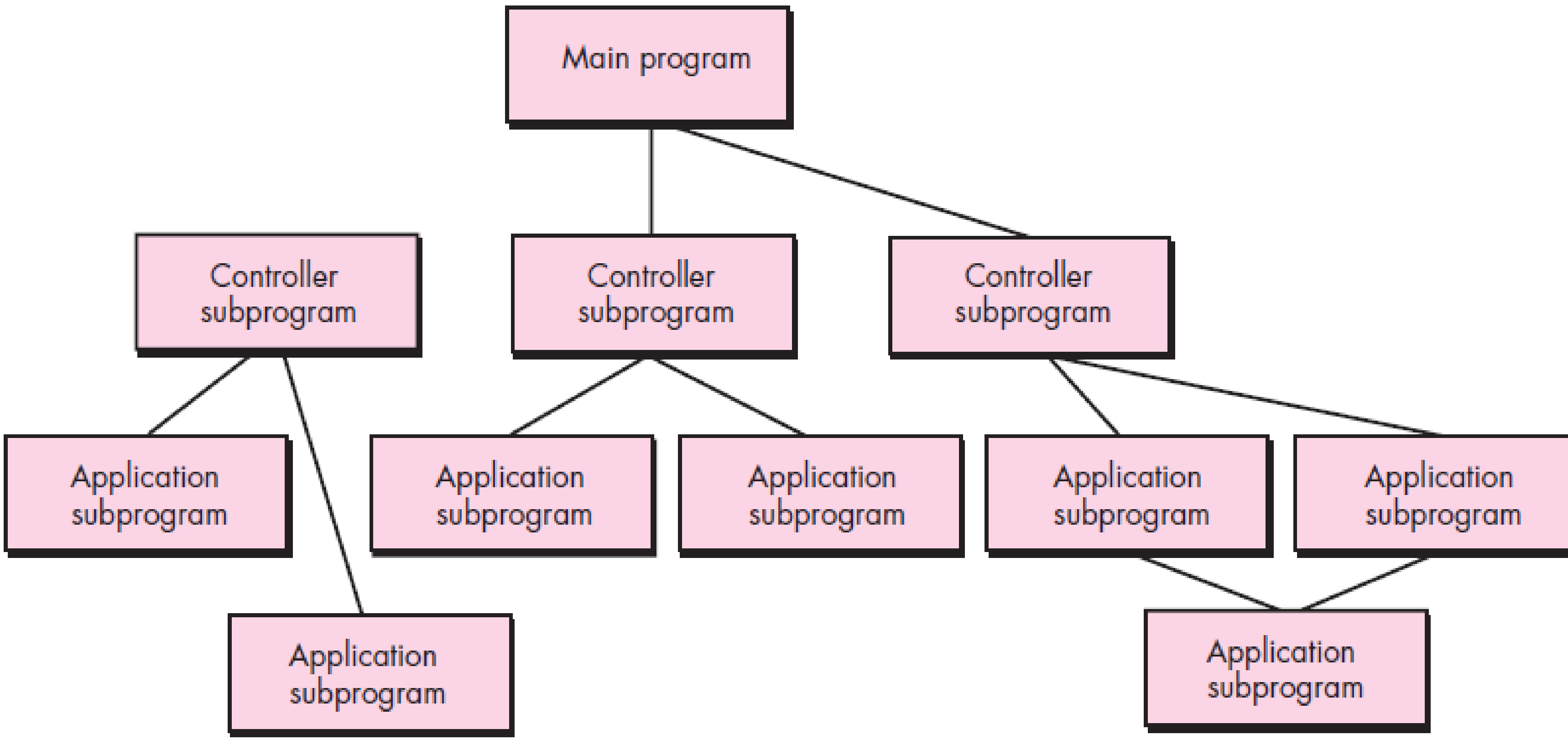


Data-flow architecture

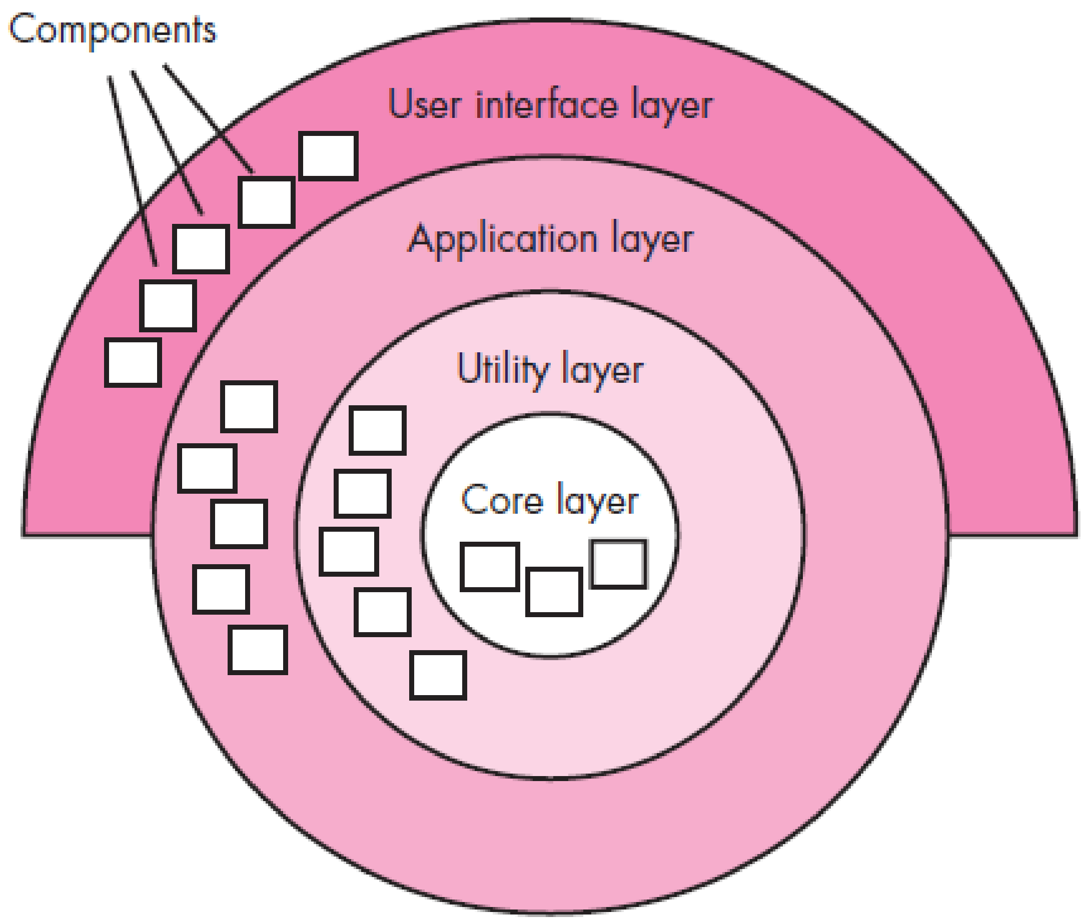


Pipes and filters

Main program / subprogram architecture



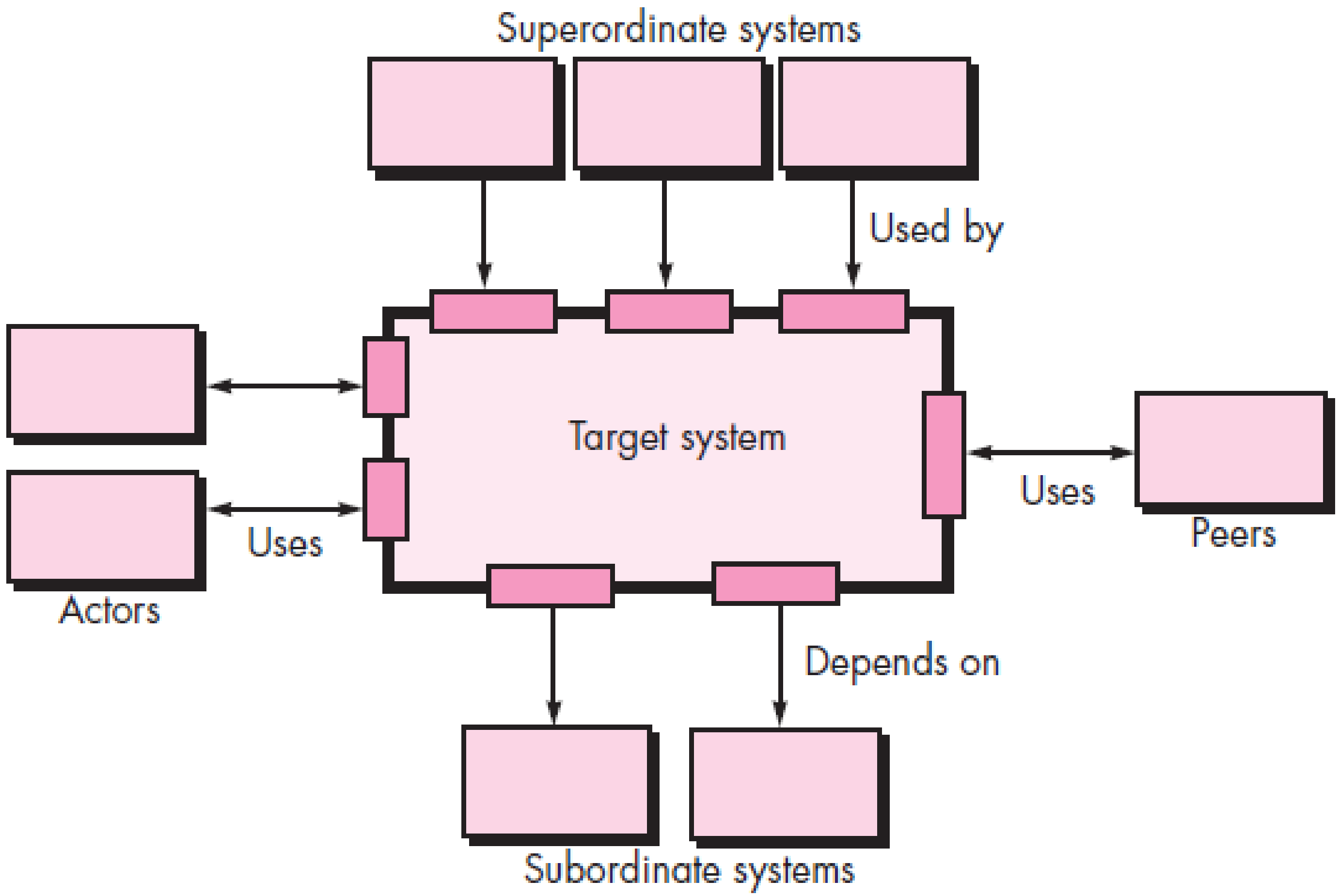
Layered architecture



Architectural Design

- The software to be developed must be put into context-that is, the design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction.

Architectural Context Diagram

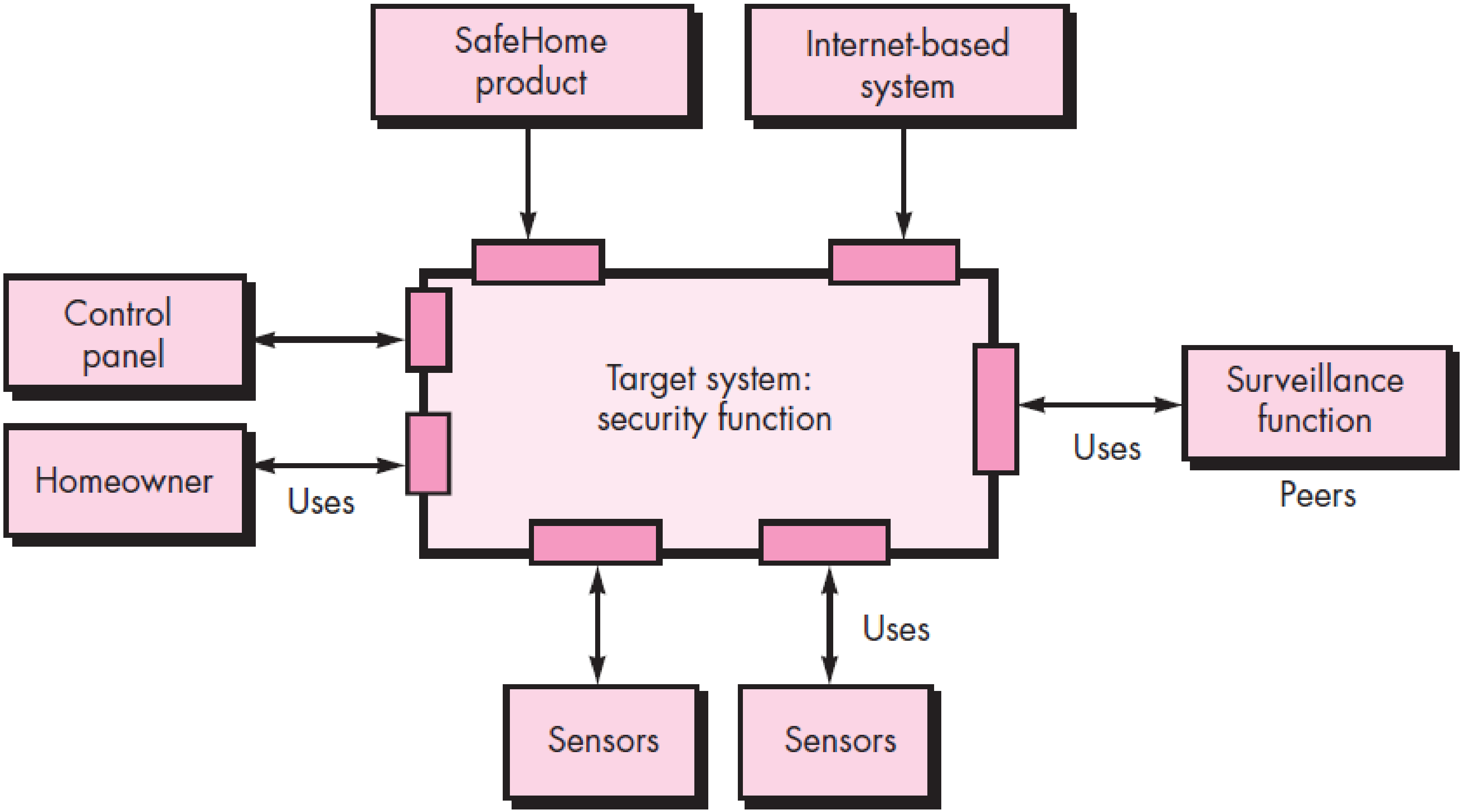


Case study

Recalling basic *SafeHome* requirements, we define four actors: **homeowner** (a user), **setup manager** (likely the same person as **homeowner,** but playing a different role), **sensors** (devices attached to the system), and the **monitoring and response subsystem** (the central station that monitors the *SafeHome* home security function). For the purposes of this example, we consider only the **homeowner** actor. The **homeowner** actor interacts with the home security function in a number of different ways using either the alarm control panel or a PC:

- Enters a password to allow all other interactions.
- Inquires about the status of a security zone.
- Inquires about the status of a sensor.
- Presses the panic button in an emergency.
- Activates/deactivates the security system.

Example


SafeHome product
Internet-based system
Control panel
Homeowner
Uses
Target system: security function
Surveillance function
Uses
Peers
Uses
Sensors
Sensors

Systems that interoperate with the *target system*

- *Superordinate systems*
  - those systems that use the target system as part of some higher-level processing scheme
- *Subordinate systems*
  - those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality
- *Peer-level systems*
  - those systems that interact on a peer-to-peer basis
- *Actors*
  - entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing

Interface Design Elements

- The interface design for software is analogous to a set of detailed drawings (and specifications) for the doors, windows, and external utilities of a house.
- These drawings depict the size and shape of doors and windows, the manner in which they operate, the way in which utility connections (e.g., water, electrical, gas, telephone) come into the house and are distributed among the rooms depicted in the floor plan.
- They tell us where the doorbell is located, whether an intercom is to be used to announce a visitor’s presence, and how a security system is to be installed.
- In essence, the detailed drawings (and specifications) for the doors, windows, and external utilities tell us how things and information flow into and out of the house and within the rooms that are part of the floor plan.

Interface Design Elements

- The interface design elements for software depict information flows into and out of the system and how it is communicated among the components defined as part of the architecture.

Elements of interface design

- the user interface (UI)
- external interfaces to other systems, devices, networks, or other producers or consumers of information
- internal interfaces between various design components.

UI design

- It is called **usability design**.
- Usability design incorporates **aesthetic elements** (e.g., layout, color, graphics, interaction mechanisms), **ergonomic elements** (e.g., information layout and placement, metaphors, UI navigation), and **technical elements** (e.g., UI patterns, reusable components).

Design of external interfaces

- The design of external interfaces requires definitive information about the entity to which information is sent or received.
- The design of external interfaces should incorporate error checking and appropriate security features.

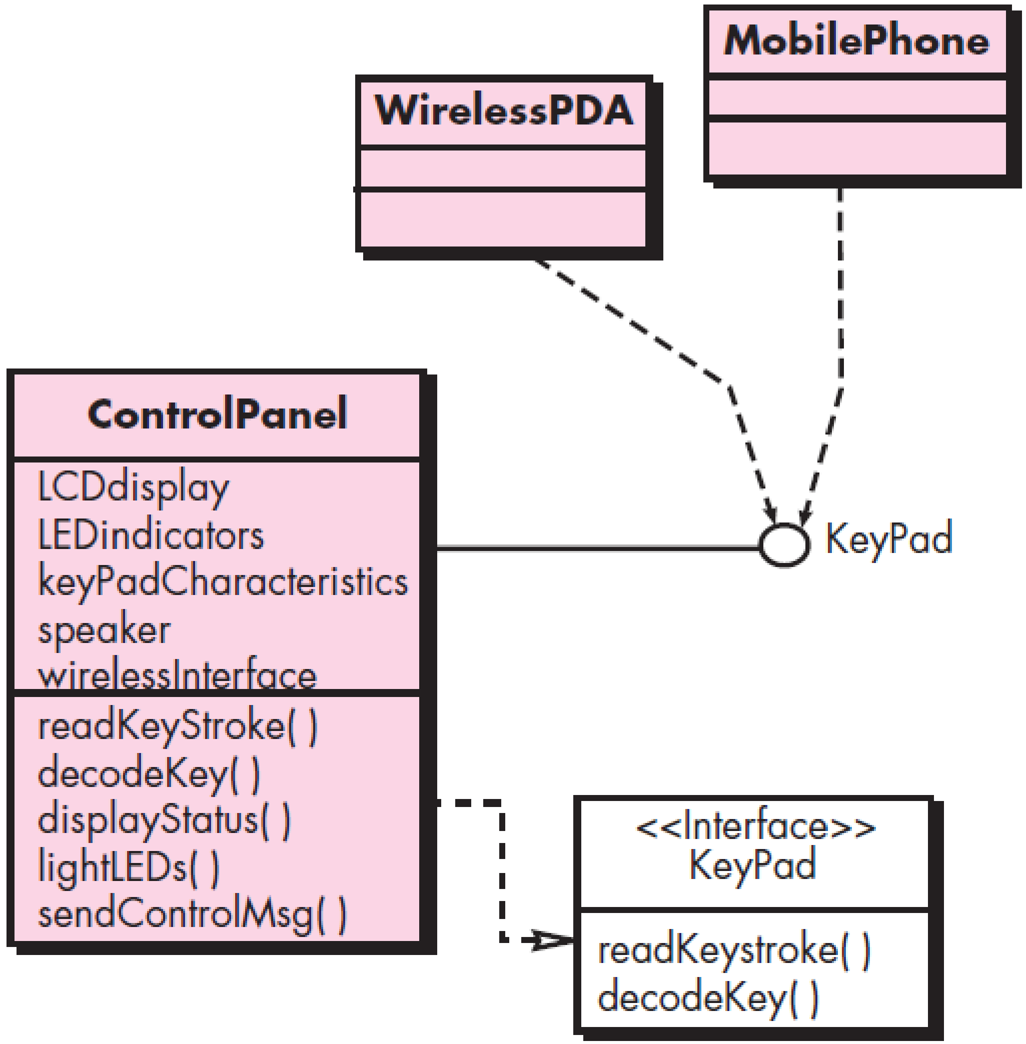
Design of internal interfaces

- The design of internal interfaces is closely aligned with **component-level design**.
- If the classic **input-process-output approach** to design is chosen, the interface of each software component is designed based on data flow representations and the functionality described in a processing narrative.

Design of internal interfaces

- In some cases, an interface is modeled in much the same way as a class.
- In UML, <u>"An interface is a specifier for the externally-visible [public] operations of a class, component, or other classifier (including subsystems) without specification of internal structure."</u>
- Stated more simply, an interface is a set of operations that describes some part of the behavior of a class and provides access to these operations.

Interface representation for Control Panel



Explanation

- The **ControlPanel class** provides the behavior associated with a keypad, and therefore, it must implement the operations *readKeyStroke () and decodeKey ().*
- If these operations are to be provided to other classes like, WirelessPDA and MobilePhone, define an interface.
- The **interface, named KeyPad**, is shown as an <<interface>> stereotype or as a small, labeled circle connected to the class with a line.
- The interface is defined with no attributes and the set of operations that are necessary to achieve the behavior of a keypad.

Explanation

- The dashed line with an open triangle at its end indicates that the ControlPanel class provides KeyPad operations as part of its behavior.
- In UML, this is characterized as a **realization**.
- That is, part of the behavior of ControlPanel will be implemented by realizing KeyPad operations. These operations will be provided to other classes that access the interface.

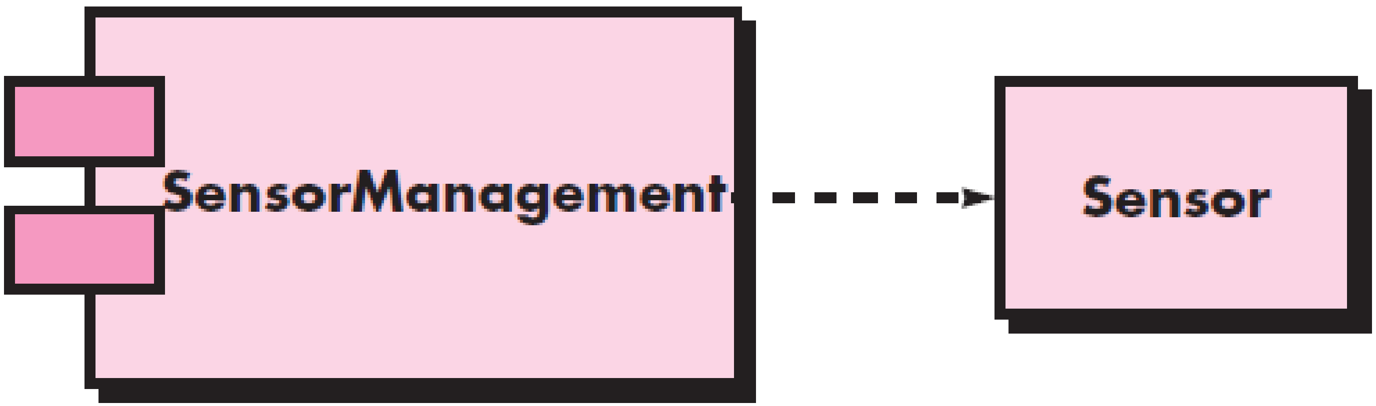
Component Level Design Elements

- The component-level design for software is the equivalent to a set of detailed drawings for each room in a house.
- These drawings depict wiring and plumbing within each room, the location of electrical receptacles and wall switches, faucets, sinks, showers, tubs, drains, cabinets, and closets.
- They also describe the flooring to be used, the moldings to be applied, and every other detail associated with a room.

Component Level Design

- The component-level design for software fully describes the internal detail of each software component.
- To accomplish this, the component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations (behaviors).

UML component Diagram



Design details of a component

- A UML activity diagram - processing logic.
- Pseudocode - Detailed procedural flow.
- Flowchart - Detailed procedural flow.
- Box diagram - Detailed procedural flow.

Example

- A component named **SensorManagement** (part of the SafeHome security function) is represented.
- A dashed arrow connects the component to a class named **Sensor** that is assigned to it.
- The SensorManagement component performs all functions associated with SafeHome sensors including monitoring and configuring them.

Deployment Level Design Elements

- Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software.
  - **Descriptor form**: the deployment diagram shows the computing environment but does not explicitly indicate configuration details.
  - **Instance form**: Each instance of the deployment (a specific, named hardware configuration) is identified.

UML deployment Diagram