



# CCA220-Analisis dan Perancangan system Informasi

[www.esaunggul.ac.id](http://www.esaunggul.ac.id)

Dosen Pengampu :

**5165-Kundang K Juman, Ir, MMSI**

Prodi Teknik Informatika dan Sistem Informasi - Fakultas  
Ilmu Komputer

# **Systems Analysis and Design**

**5th Edition**

## **Chapter 10. Program Design**

**Alan Dennis, Barbara Haley Wixom, and Roberta Roth**

# Chapter 10 Outline

---

- Moving from logical to physical process models.
- Designing programs.
- Structure chart.
- Program specification.

# INTRODUCTION

---

- **Program design** – Analysts determine what programs will be written and create instructions for the programmers.
- Various implementation decisions are made about the new system (e.g., what programming language(s) will be used.)
- The DFDs created during analysis are modified to show these implementation decisions, resulting in a set of *physical data flow diagrams*.
- The analysts determine how the processes are organized, using a *structure chart* to depict the decisions.
- Detailed instructions called *program specifications* are developed.

# MOVING FROM LOGICAL TO PHYSICAL PROCESS MODELS

---

- The analysis phase focuses on logical data flow diagrams which do not contain any indication how the system will actually be implemented; they simply state what the new system will do.
- During design, *physical process models* are created to show implementation details and explain how the final system will work.

# The Physical Data Flow Diagram

---

- The physical DFD contains the same components as the logical DFD, and the same rules apply.
- The basic difference between the two models is that a physical DFD contains additional details that describe how the system will be built.
- There are five steps to perform to make the transition to the physical DFD.

# Transition from logical to physical DFD by:

---

- Adding implementation references
- Add human-machine boundary
- Add system-related data stores, data flows, and processes
- Update data elements in data flows
- Update metadata in CASE repository

# (cont'd)

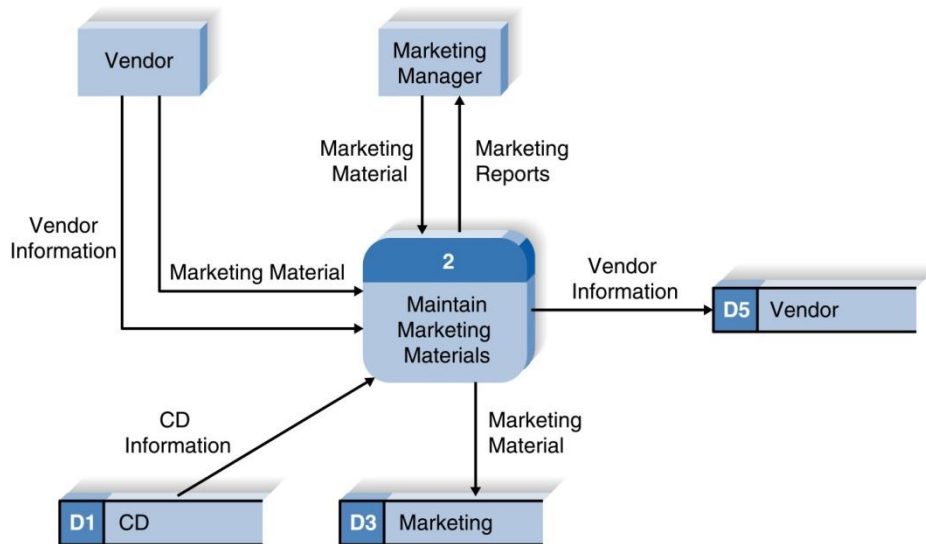
Step	Explanation
Add implementation references	Using the existing logical DFD, place the way in which the data stores, data flows, and processes will be implemented in parentheses below each component.
Draw a human-machine boundary	Draw a line to separate the automated parts of the system from the manual parts.
Add system-related data stores, data flows, and processes	Add system-related data stores, data flows, and processes to the model (components that have little to do with the business process).
Update the data elements in the data flows	Update the data flows to include system-related data elements.
Update the metadata in the CASE repository	Update the metadata in the CASE repository to include physical characteristics.

CASE = computer-aided software engineering; DFD = data flow diagram.

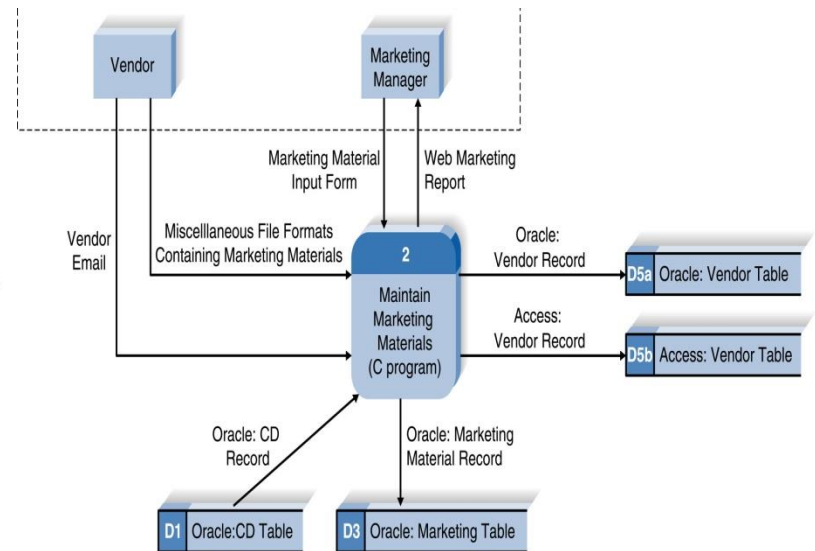


# Contrasting Logical and Physical DFDs

## A Logical DFD



## A Physical DFD



# (cont'd)

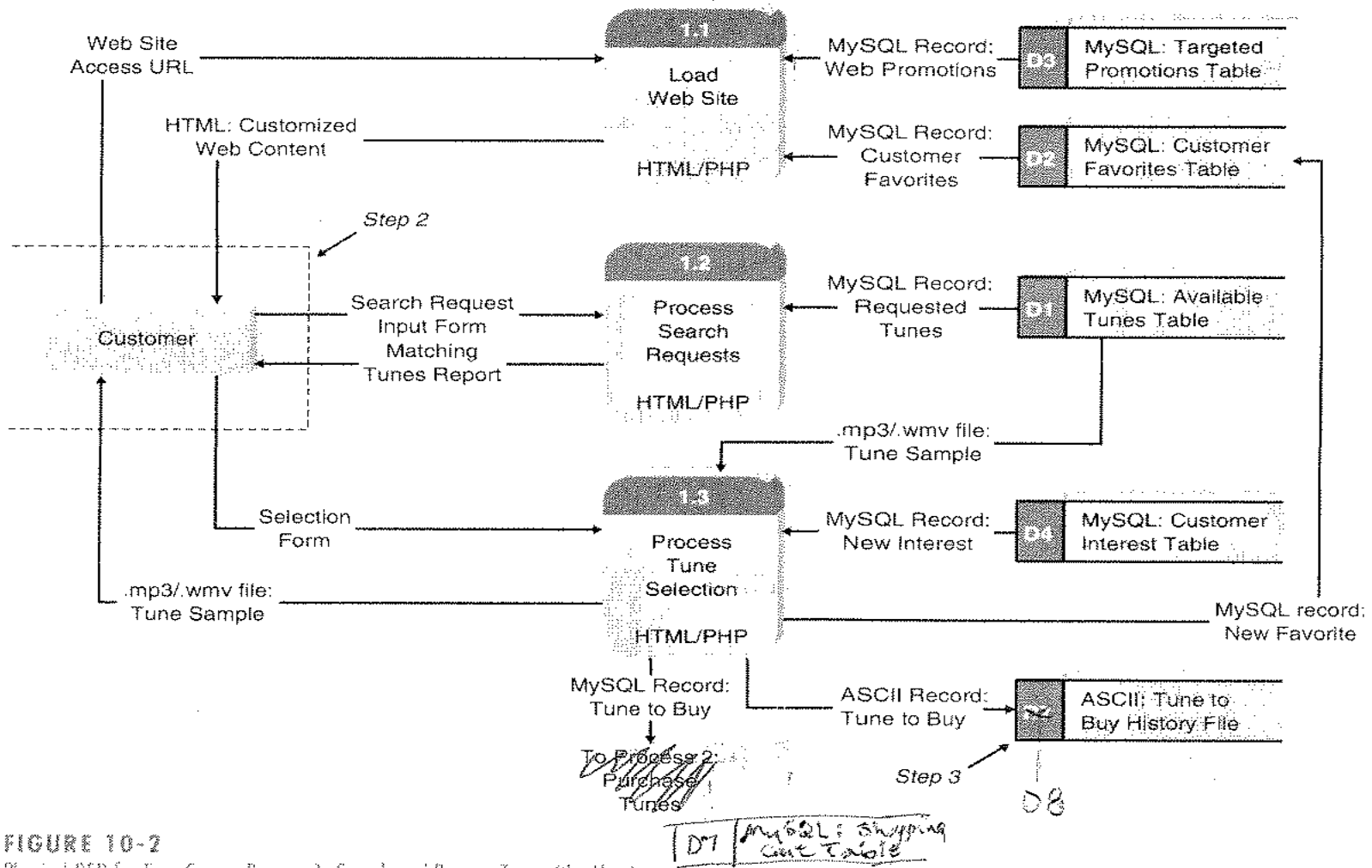


FIGURE 10-2

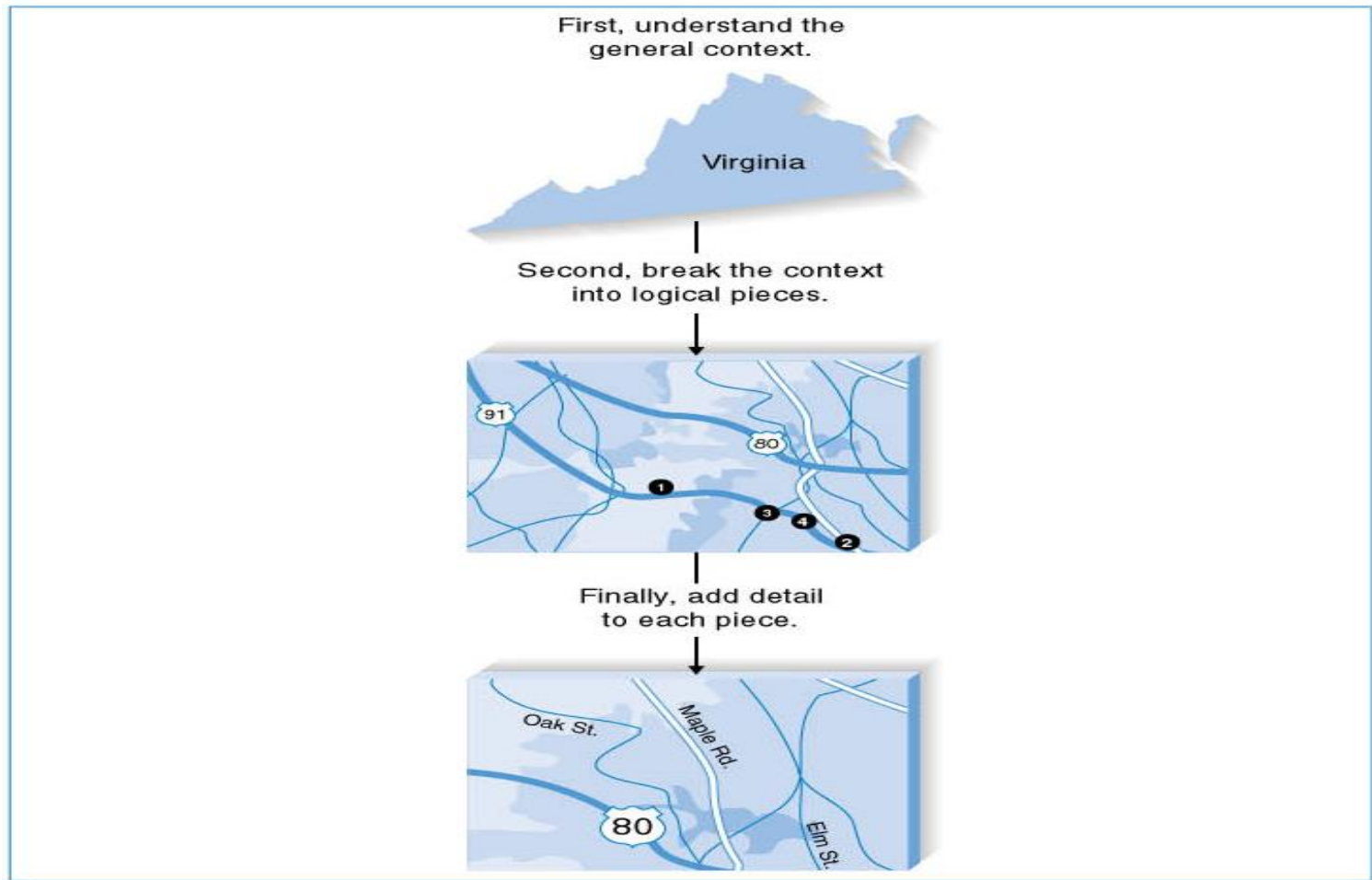
Physical DFD for Tune Source Process 1: Search and Browse Tunes (the How)

# DESIGNING PROGRAMS

---

- Analysts should create a design of a maintainable system that is modular and flexible.
- Analysts can design programs in a *top-down modular approach*.

# (cont'd)



# (cont'd)

---

- Program design in the top-down modular approach:
  - A high-level diagram, called **structure chart**, is created to illustrate the organization and interaction of the different pieces of code within the program.
  - Program specifications are written to describe what needs to be included in each program module.
- At the end of program design, the project team compiles the **program design** document.

# STRUCTURE CHART

---

- The ***structure chart*** is an Important program design technique that help the analyst design the program.
- It shows all components of code in a hierarchical format that implies
  - ***sequence***
    -
  - ***selection***
    -
  - ***Iteration***
    -

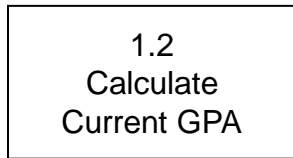
# Structure chart elements

---

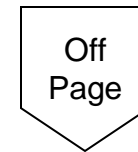
- Modules
  - Control modules
  - Subordinate modules
  - Library modules
- Connecting lines
  - Curved arrow (loop)
  - Conditional line (diamond)
- Connector
  - Off-page
  - On-page
- Couples
  - Data couple
  - Control couple

# Structure Chart Elements

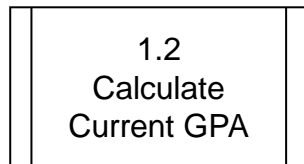
Module



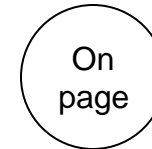
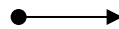
Conditional Line



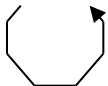
Library Module



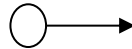
Control Couple



Loop

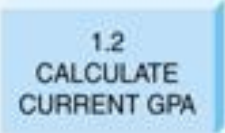





Data Couple







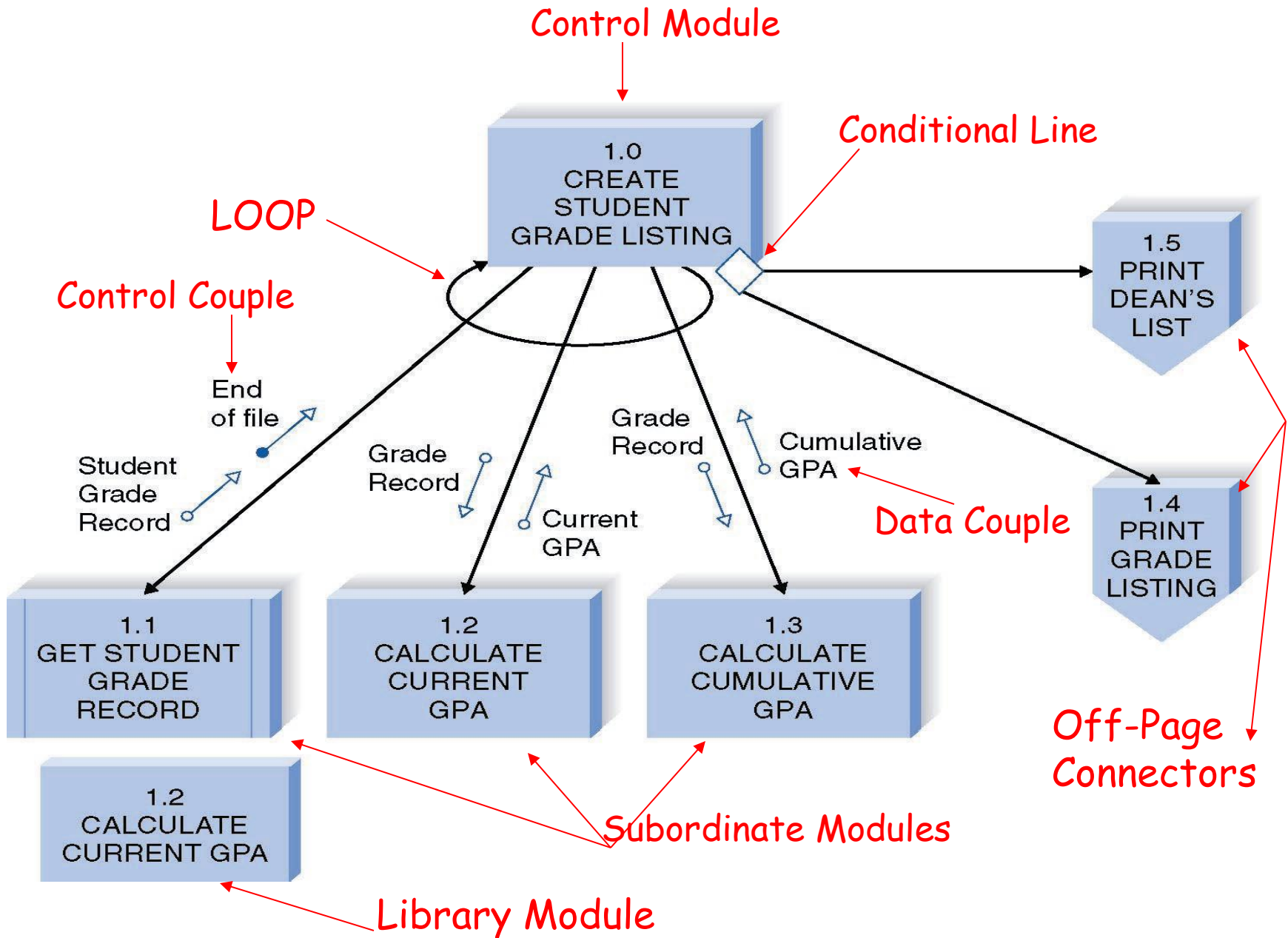


# Syntax of Structure Chart

Structure Chart Element	Purpose	Symbol
<p>Every <i>module</i>:</p> <ul style="list-style-type: none"><li>• Has a number.</li><li>• Has a name.</li><li>• Is a control module if it calls other modules below it.</li><li>• Is a subordinate module if it is controlled by a module at a higher level.</li></ul>	Denotes a logical piece of the program	
<p>Every <i>library module</i> has:</p> <ul style="list-style-type: none"><li>• A number.</li><li>• A name.</li><li>• Multiple instances within a diagram.</li></ul>	Denotes a logical piece of the program that is repeated within the structure chart	
<p>A <i>loop</i>:</p> <ul style="list-style-type: none"><li>• Is drawn with a curved arrow.</li><li>• Is placed around lines of one or more modules that are repeated.</li></ul>	Communicates that a module(s) is repeated	
<p>A <i>conditional line</i>:</p> <ul style="list-style-type: none"><li>• Is drawn with a diamond.</li><li>• Includes modules that are invoked on the basis of some condition.</li></ul>	Communicates that subordinate modules are invoked by the control module based on some condition	

# (cont'd)

<p><i>A data couple:</i></p> <ul style="list-style-type: none"><li>• Contains an arrow.</li><li>• Contains an empty circle.</li><li>• Names the type of data that are being passed.</li><li>• Can be passed up or down.</li><li>• Has a direction that is denoted by the arrow.</li></ul>	Communicates that data are being passed from one module to another	
<p><i>A control couple:</i></p> <ul style="list-style-type: none"><li>• Contains an arrow.</li><li>• Contains a filled-in circle.</li><li>• Names the message or flag that is being passed.</li><li>• Should be passed up, not down.</li><li>• Has a direction that is denoted by the arrow.</li></ul>	Communicates that a message or a system flag is being passed from one module to another	
<p><i>An off-page connector:</i></p> <ul style="list-style-type: none"><li>• Is denoted by the hexagon.</li><li>• Has a title.</li><li>• Is used when the diagram is too large to fit everything on the same page.</li></ul>	Identifies when parts of the diagram are continued on another page of the structure chart	
<p><i>An on-page connector:</i></p> <ul style="list-style-type: none"><li>• Is denoted by the circle.</li><li>• Has a title.</li><li>• Is used when the diagram is too large to fit everything in the same spot on a page.</li></ul>	Identifies when parts of the diagram are continued somewhere else on the same page of the structure chart	

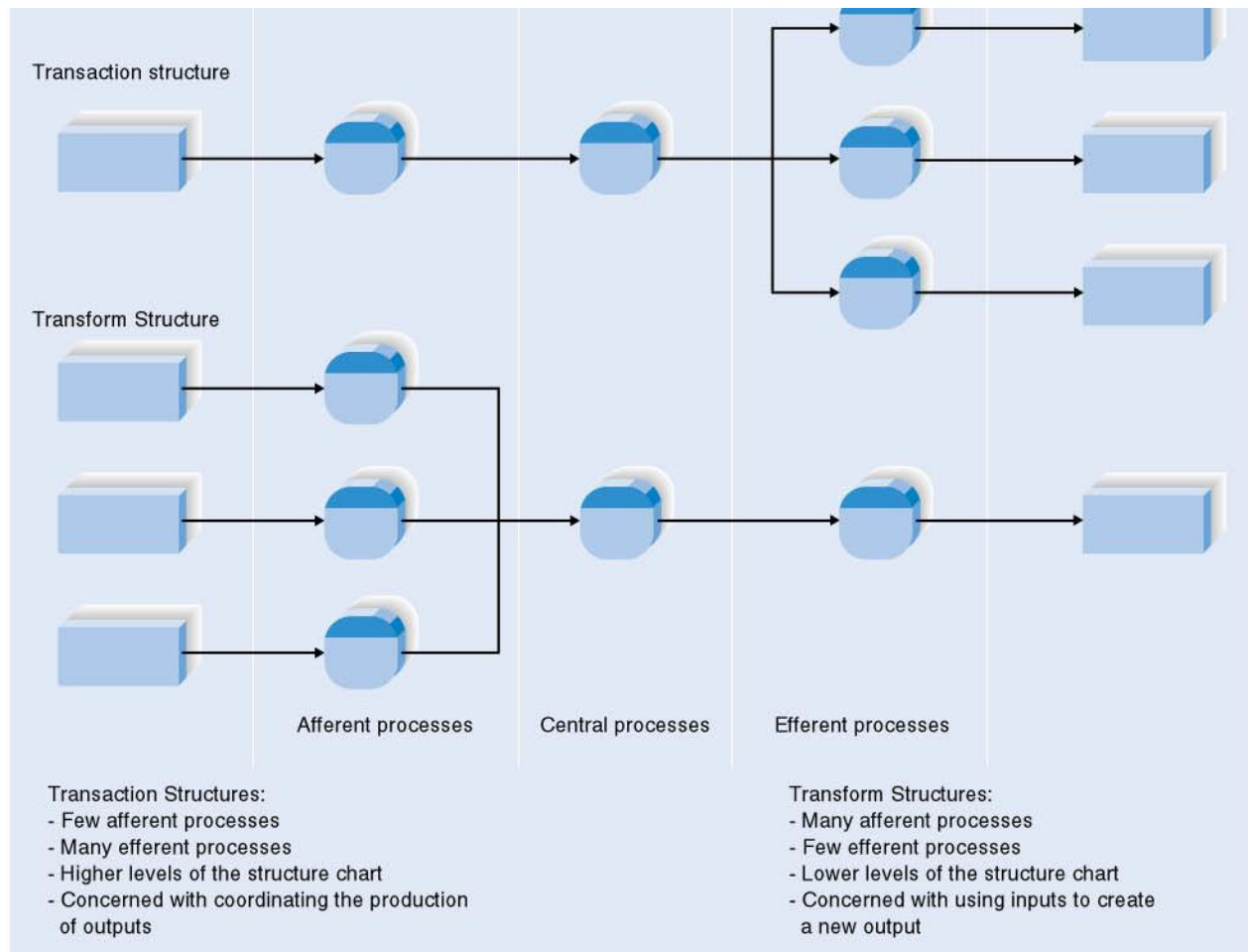


# Building the Structure Chart

---

- Process models are used as the start point for structure charts.
- There are three basic kinds of processes:
  - *Afferent processes* provide inputs to system,
  - *Central processes* perform critical functions in the operation, and
  - *Efferent processes* deal with system outputs.
- Each DFD level tends to correspond to a different level of the structure chart hierarchy.

# (cont'd)



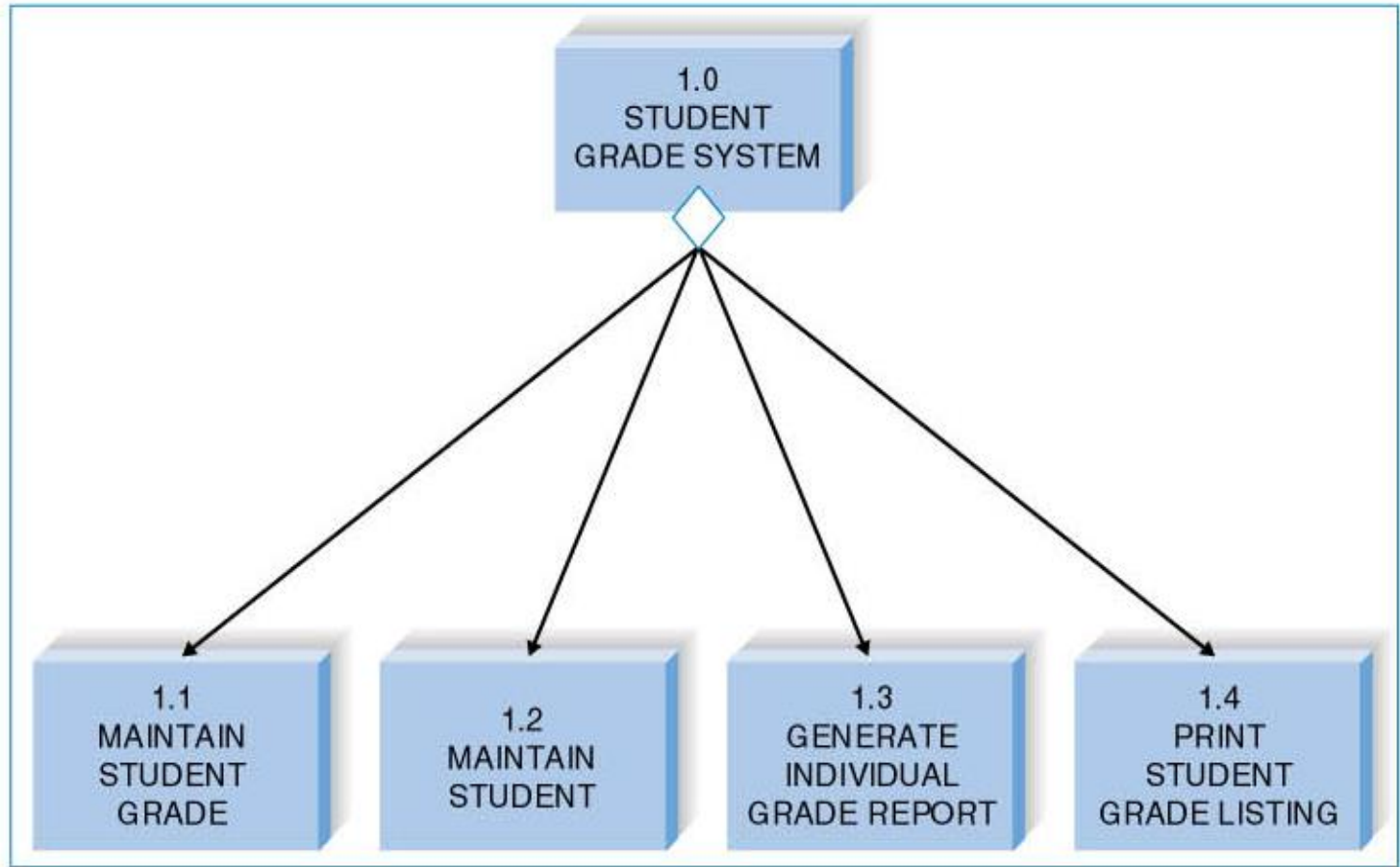
# Transaction Structure

---

- There are two basic arrangement, or structures, for combining modules.
- The ***transaction structure*** contains a control module that calls subordinate modules, each of which handles a particular transaction.
  - Few afferent processes
  - Many efferent processes
  - Higher levels of the structure chart
  - Connected with coordinating the production of outputs

# (cont'd)

---



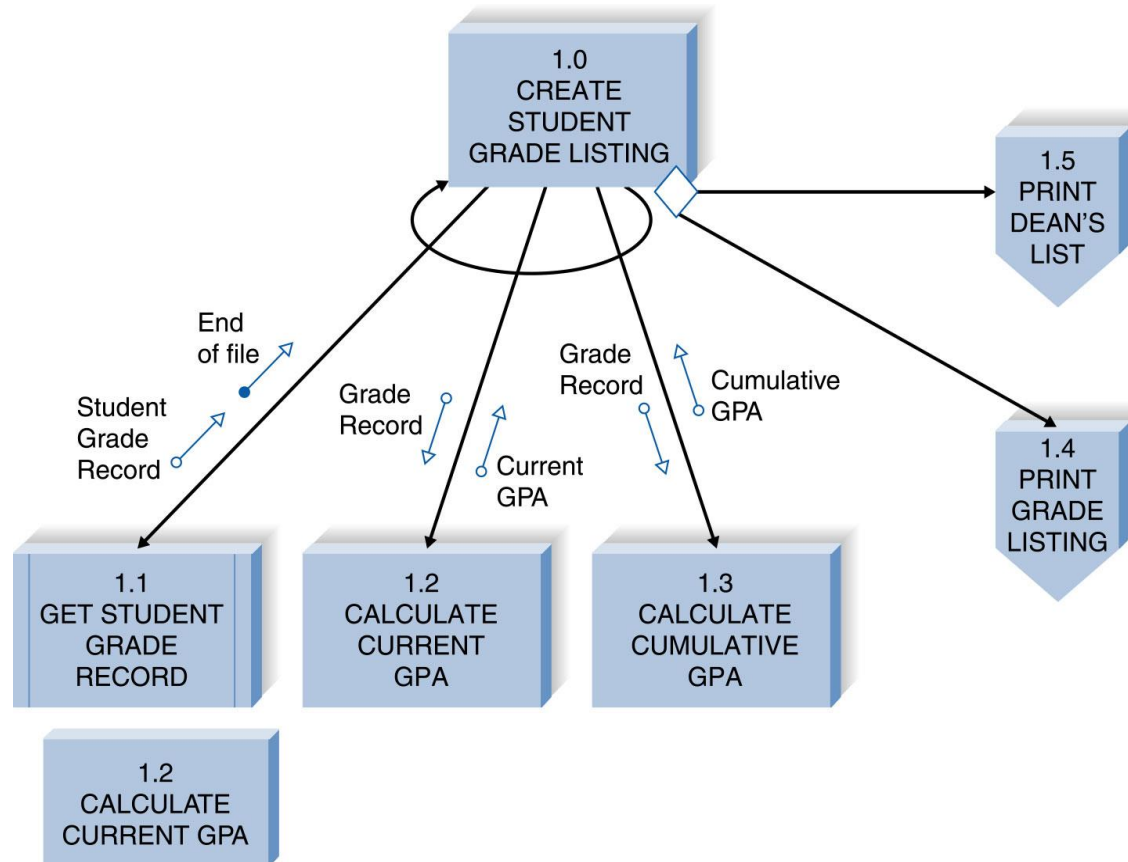
# Transform Structure

---

- The *transaction structure* has a control module that calls several subordinate modules in sequence after which something “happens.”
- These modules are related because together they form a process that transforms some input into an output.
  - Many afferent processes
  - Few efferent processes
  - Lower levels of the structure chart
  - Connected with using inputs to create a new output



# Transform Structure



# Steps in Building the Structure Chart

---

- Identify top level modules and decompose them into lower levels.
  -
- Identify special connections.
  -
- Add couples.
  -
- Revise structure chart.
  -

# Design Guidelines

---

- High quality structure charts result in programs that are modular, reusable, and easy to implement.
- Measures of good design include
  - cohesion,
  - coupling, and
  - appropriate levels of fan-in and fan-out.

# Build Modules with High Cohesion

---

- ***Cohesion*** refers to how well the lines of code within each module relate to each other.
- Cohesive modules are easy to understand and build because their code performs one function effectively.

# (cont'd)

---

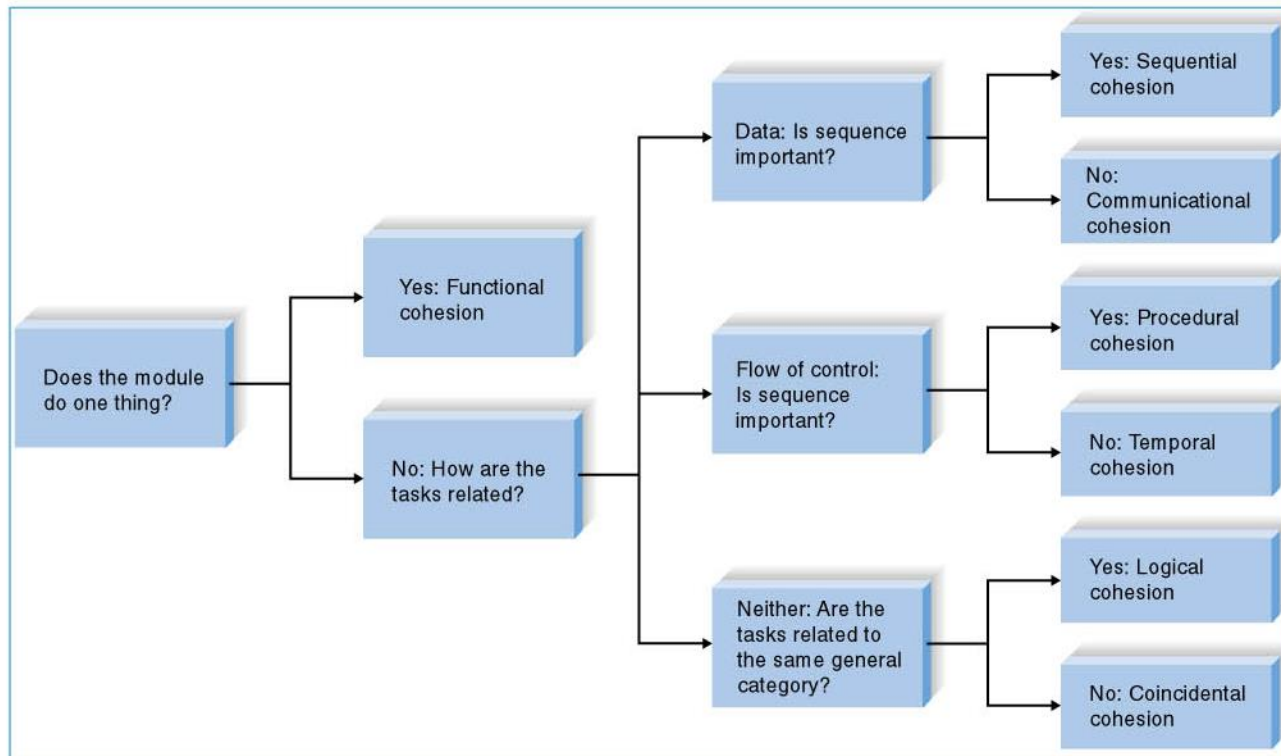
- There are various types of cohesion.
- ***Functional cohesion*** – all elements of the modules contribute to performing a single task.
- ***Temporal cohesion*** – functions are invoked at the same time.
- ***Coincidental cohesion*** – there is no apparent relationship among a module's functions.

# (cont'd)

Type		Definition	Example
Good ↓ Bad	Functional	Module performs one problem-related task	<div style="border: 1px solid black; padding: 2px; display: inline-block;">Calculate Current GPA</div> The module calculates current GPA only
	Sequential	Output from one task is used by the next	<div style="border: 1px solid black; padding: 2px; display: inline-block;">Format and Validate Current GPA</div> Two tasks are performed, and the formatted GPA from the first task is the input for the second task
	Communicational	Elements contribute to activities that use the same inputs or outputs	<div style="border: 1px solid black; padding: 2px; display: inline-block;">Calculate Current and Cumulative GPA</div> Two tasks are performed because they both use the student grade record as input
	Procedural	Elements are performed in sequence but do not share data	<div style="border: 1px solid black; padding: 2px; display: inline-block;">Print Grade Listing</div> The module includes the following: housekeeping, produce report
	Temporal	Activities are related in time	<div style="border: 1px solid black; padding: 2px; display: inline-block;">Initialize Program Variables</div> Although the tasks occur at the same time, each task is unrelated
	Logical	List of activities; which one to perform is chosen outside of module	<div style="border: 1px solid black; padding: 2px; display: inline-block;">Perform Customer Transaction</div> This module will open a checking account, open a savings account, or calculate a loan, depending on the message that is sent by its control module
	Coincidental	No apparent relationship	<div style="border: 1px solid black; padding: 2px; display: inline-block;">Perform Activities</div> This module performs different functions that have nothing to do with each other: update customer record, calculate loan payment, print exception report, analyze competitor pricing structure

# (cont'd)

## ■ Cohesion Decision Tree



## (cont'd)

---

- ***Factoring*** is the process of separating out a function from one module into a module of its own.
- Factoring can make modules cohesive create a better structure.

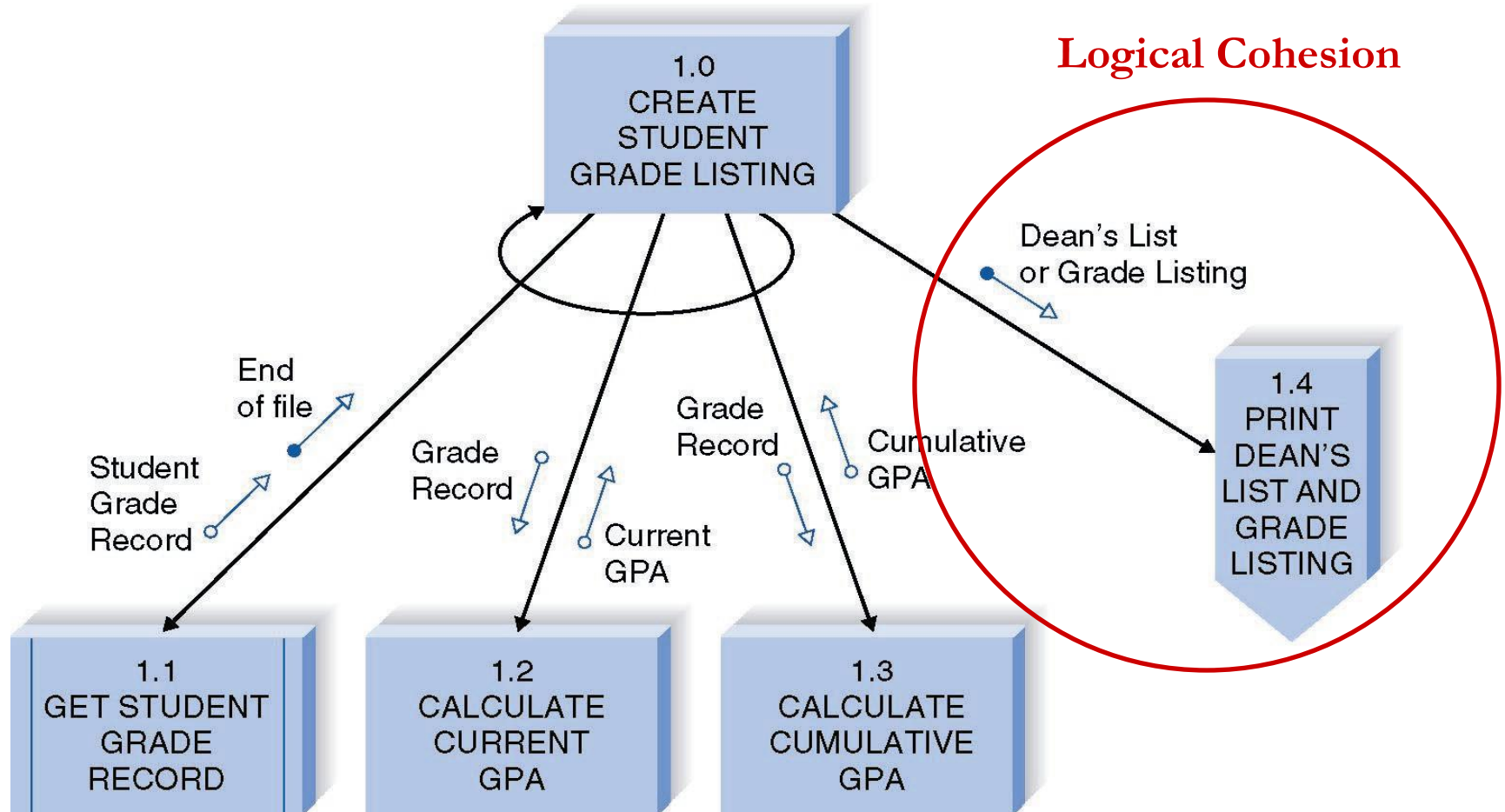


# Build Loosely Coupled Modules

---

- ***Coupling*** involves how closely modules are interrelated.
- Modules should be loosely coupled.
- ***Data coupling*** occurs when modules pass parameters or specific data to each other. It is preferred.
- ***Content coupling*** occurs when one module actually refers to the inside of another module. It is a bad coupling type.

# Example of Low Cohesion



# Create High Fan-In

---

- ***Fan-in*** describes the number of control modules that communicate with a subordinate.
- A module with high fan-in has many different control modules that call it. This is a good situation because high fan-in indicates that a module is reused in many places on the structure chart.

# Avoid High Fan-Out

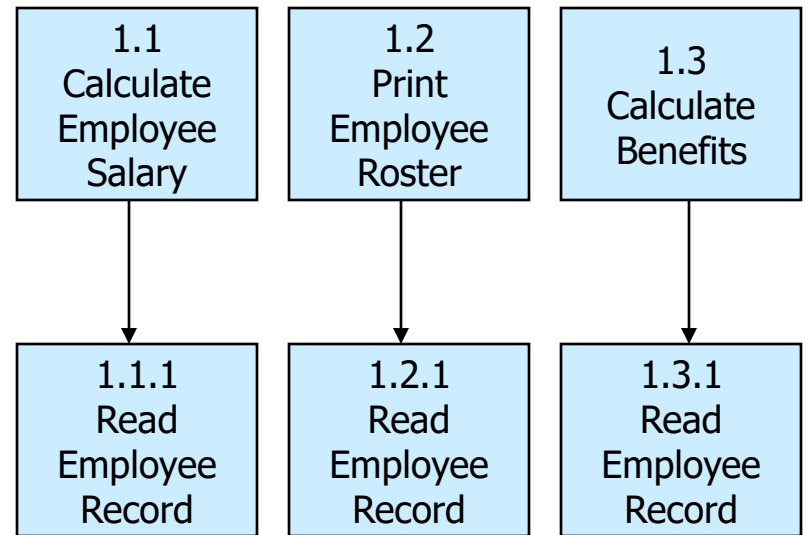
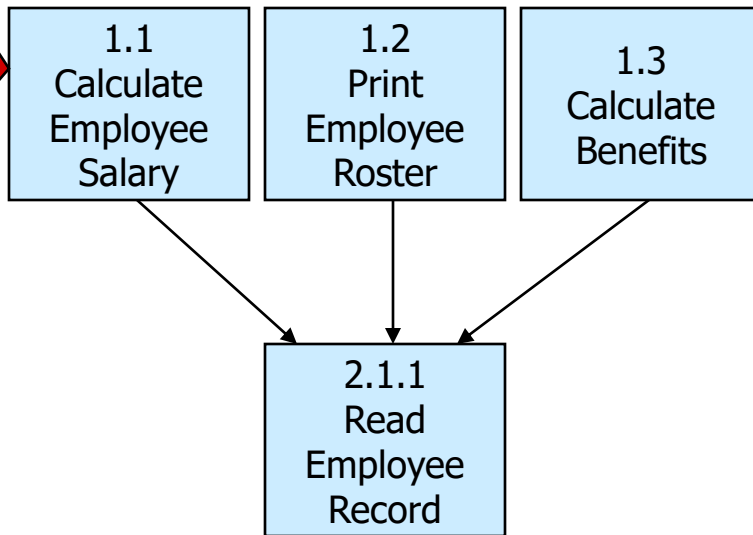
---

- A large number of subordinates associated with a single control should be avoided.
- The general rule of thumb is to limit a control module's subordinates to approximately **seven** for low ***fan-out***.

# Fan-in

High fan-in preferred

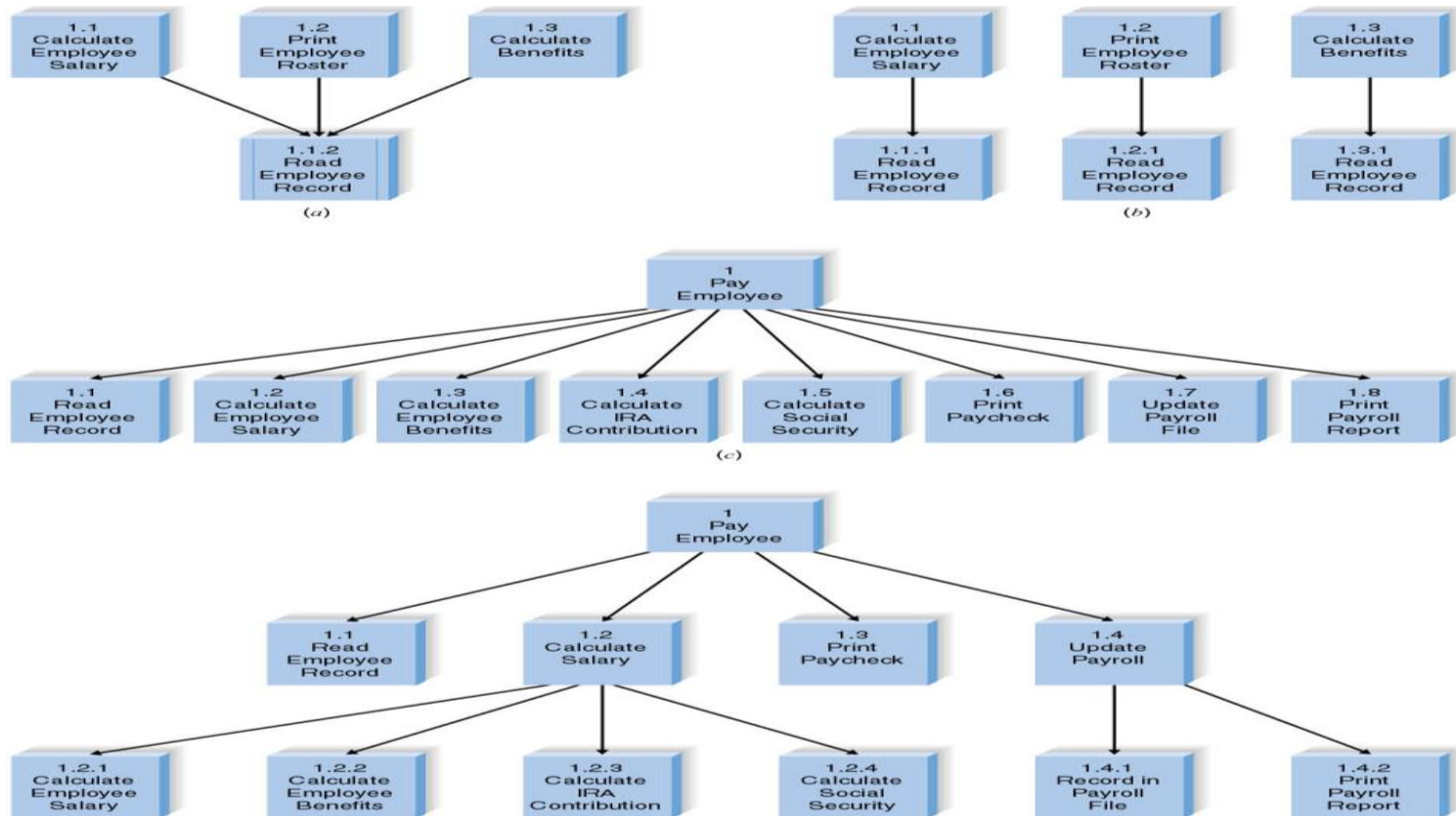
Promotes reuse of subordinate modules



Low fan-in not preferred

# (cont'd)

## Examples of Fan-in and Fan-out



# Assess the Chart for Quality

---

## ■ Check list for structure chart quality



- ✓ Library modules have been created whenever possible.
- ✓ The diagram has a high fan-in structure.
- ✓ Control modules have no more than seven subordinates.
- ✓ Each module performs only one function (high cohesion).
- ✓ Modules sparingly share information (loose coupling).
- ✓ Data couples that are passed are actually used by the accepting module.
- ✓ Control couples are passed from “low to high.”
- ✓ Each module has a reasonable amount of code associated with it.

# PROGRAM SPECIFICATION

---

- ***Program Specifications*** are documents that include explicit instructions on how to program pieces of code.
- There is no formal syntax for program specification.
- Four components are essential for program specification:
  - Program information;
  - Events;
  - Inputs and outputs; and
  - ***Pseudocode*** – a detailed outline of lines of code that need to be written.
- Additional notes and comments can also be included.



# (cont'd)

## Program specification form

Program Specification 1.1 for ABC System

Module \_\_\_\_\_  
Name: \_\_\_\_\_  
Purpose: \_\_\_\_\_  
Programmer: \_\_\_\_\_  
Date due: \_\_\_\_\_

C                      PowerScript                      COBOL                      Visual Basic

Events \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

Input Name	Type	Used By	Notes

Output Name	Type	Used By	Notes

Pseudocode \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

Other \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

# (cont'd)

---

## Pseudocode Example

```
(Get_CD_Info module)  
  Accept (CD.Title) {Required}  
  Accept (CD.Artist) {Required}  
  Accept (CD.Category) {Required}  
  Accept (CD.Length)  
Return
```

# SUMMARY

---

- **Moving from logical to physical process models.**
  - **Physical DFDs** show implementation details.
- **Structure chart.**
  - The structure chart shows all of the functional components needed in the program at a high level.
- **Building structure chart.**
  - Module, control connection, couples, review.
- **Structure chart design guidelines.**
  - Cohesion, coupling, and fan-in/fan-out.
- **Program specifications.**
  - Program specifications provide more detailed instructions to the programmers.

## **Copyright 2011 John Wiley & Sons, Inc.**

---

All rights reserved. Reproduction or translation of this work beyond that permitted in Section 117 of the 1976 United States Copyright Act without the express written permission of the copyright owner is unlawful. Request for further information should be addressed to the Permissions Department, John Wiley & Sons, Inc. The purchaser may make back-up copies for his/her own use only and not for redistribution or resale. The Publisher assumes no responsibility for errors, omissions, or damages, caused by the use of these programs or from the use of the information contained herein.