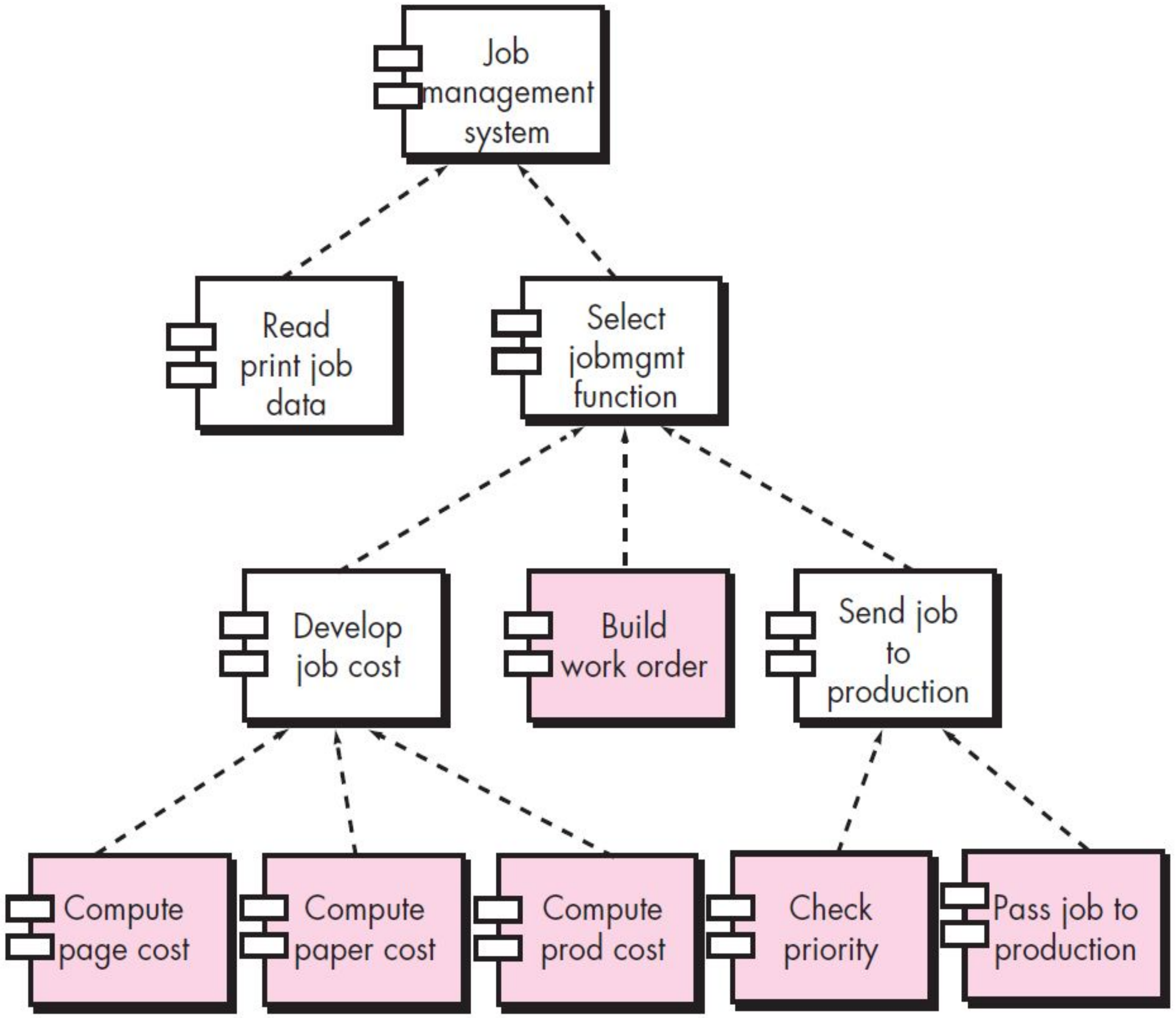
Component-Level Design

Ashish Kumar Dwivedi

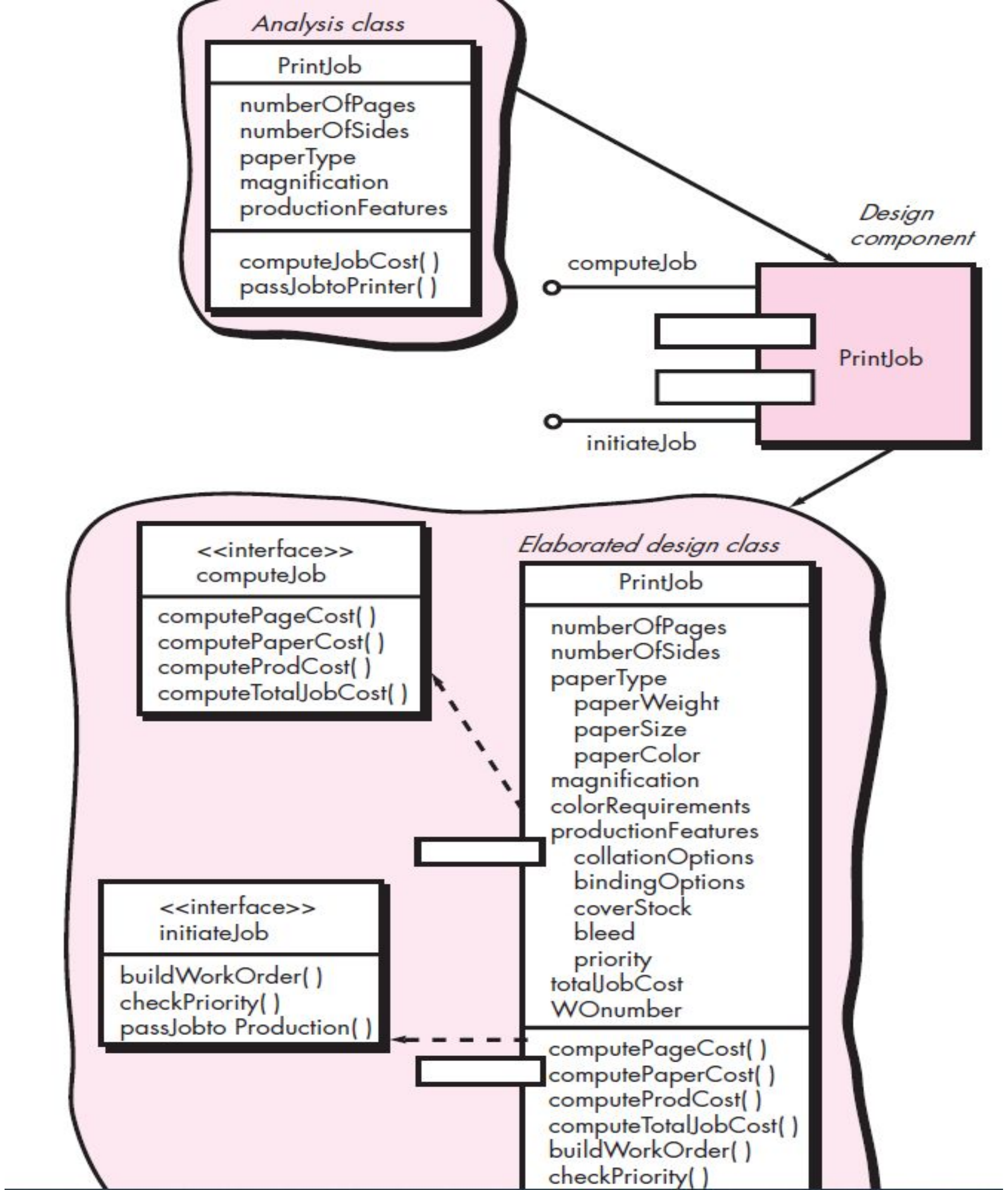
Component-level design

- Component-level design occurs after the first iteration of architectural design has been completed.
- At this stage, the overall data and program structure of the software has been established.
- The intent is to translate the design model into operational software.
- But the level of abstraction of the existing design model is relatively high, and the abstraction level of the operational program is low.
- **A *component* is a modular building block for computer software.**
- **a component as: ". . . a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."**

Structure Chart for traditional system



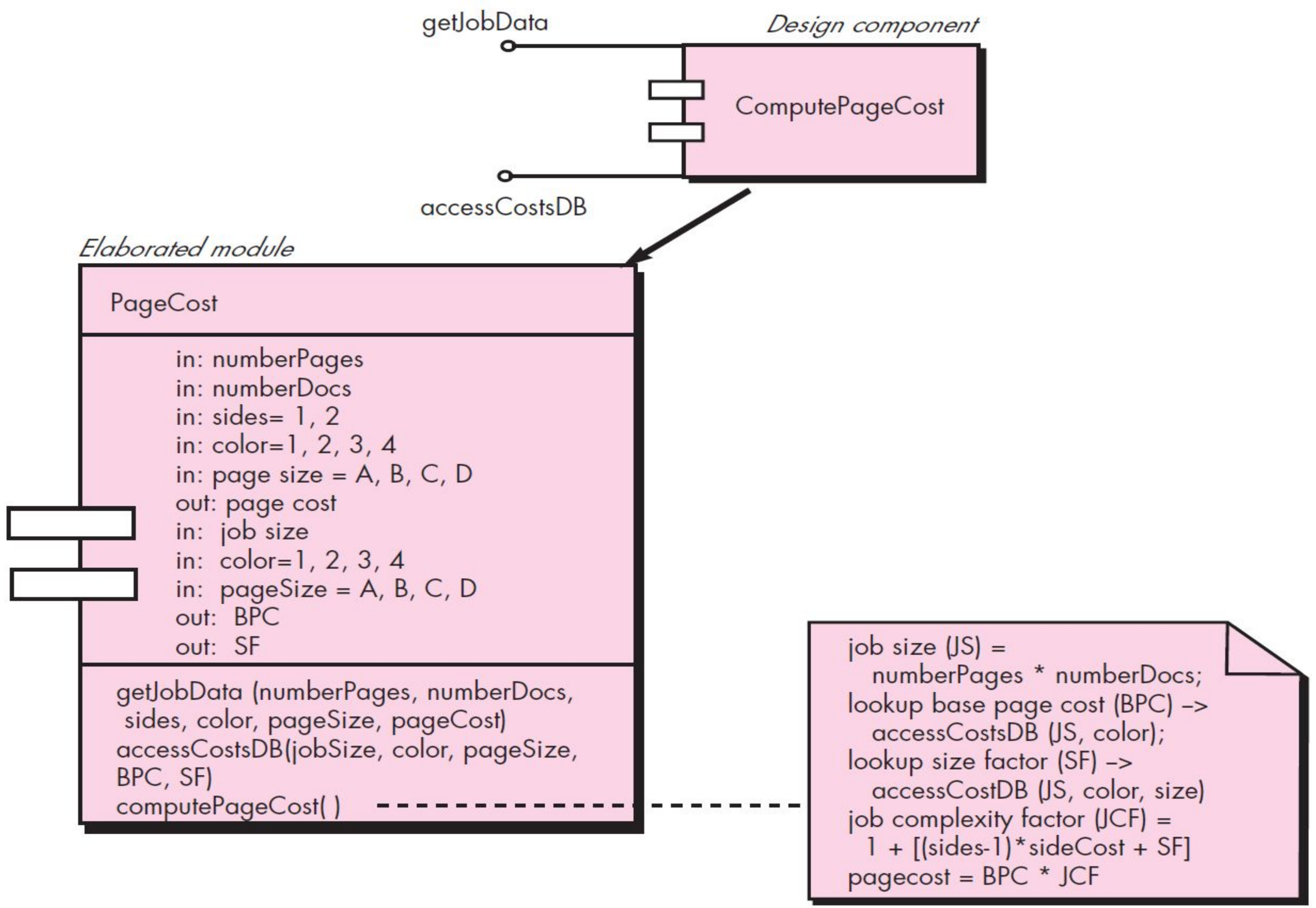
An Object-Oriented View



Module *ComputePageCost*

- Data required to perform this function are:
  - number of pages in the document,
  - total number of documents to be produced,
  - one- or two-side printing, color requirements,
  - and size requirements.
- These data are passed to *ComputePageCost* via the module's interface.
- Page cost is inversely proportional to the size of the job and directly proportional to the complexity of the job.

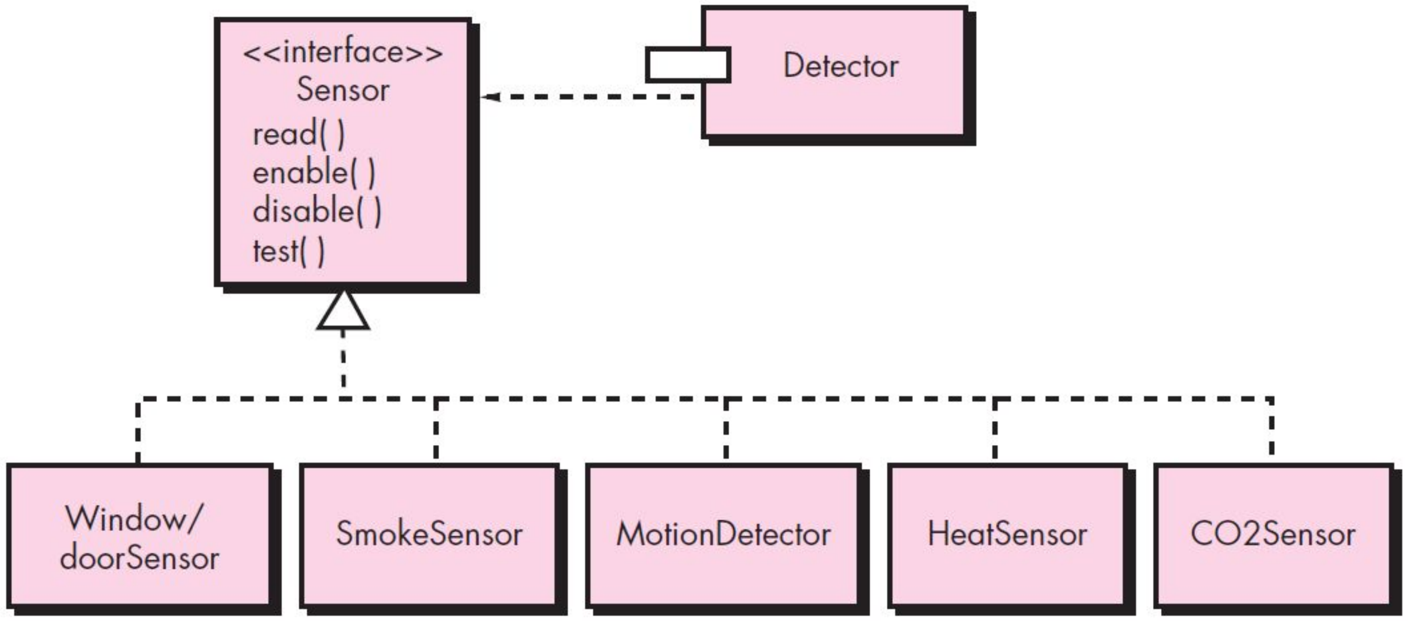
- The *ComputePageCost* module accesses data by invoking the module *getJobData,* which allows all relevant data to be passed to the component, and a database interface, *accessCostsDB,* which enables the module to access a database that contains all printing costs.
- As design continues, the *ComputePageCost* module is elaborated to provide algorithm detail and interface detail (in above figure).
- Algorithm detail can be represented using the pseudocode text shown in the figure or with a UML activity diagram.
- The interfaces are represented as a collection of input and output data objects or items.
- Design elaboration continues until sufficient detail is provided to guide construction of the component.


getJobData
Design component
ComputePageCost
accessCostsDB
Elaborated module
PageCost
in: numberPages
in: numberDocs
in: sides= 1, 2
in: color=1, 2, 3, 4
in: page size = A, B, C, D
out: page cost
in: job size
in: color=1, 2, 3, 4
in: pageSize = A, B, C, D
out: BPC
out: SF
getJobData (numberPages, numberDocs, sides, color, pageSize, pageCost)
accessCostsDB(jobSize, color, pageSize, BPC, SF)
computePageCost( )
job size (JS) = numberPages * numberDocs;
lookup base page cost (BPC) –> accessCostsDB (JS, color);
lookup size factor (SF) –> accessCostDB (JS, color, size)
job complexity factor (JCF) = 1 + [(sides-1)*sideCost + SF]
pagecost = BPC * JCF

Basic Design principles: used in class-based component

1. **The Open-Closed Principle (OCP).** *"A module [component] should be open for extension but closed for modification"*

- The *sensor* interface presents a consistent view of sensors to the detector component. If a new type of sensor is added no change is required for the **Detector** class (component). The OCP is preserved.



2. **The Liskov Substitution Principle (LSP).**

- *“Subclasses should be substitutable for their base classes”*
- ***Example:*** A “contract” is a *precondition* that must be true before the component uses a base class and a *postcondition* that should be true after the component uses a base class. When you create derived classes, be sure they conform to the pre- and postconditions.

3. **Dependency Inversion Principle (DIP).**

- *“Depend on abstractions. Do not depend on concretions”*
- As we have seen in the discussion of the OCP, abstractions are the place where a design can be extended without great complication.
- The more a component depends on other concrete components (rather than on abstractions such as an interface), the more difficult it will be to extend.

4. **The Interface Segregation Principle (ISP).**

- *“Many client-specific interfaces are better than one general purpose interface”*
- Many instances in which multiple client components use operations provided by a server class.
- ISP suggests, you should create a specialized interface to serve each major category of clients.
- **Example:** consider the **FloorPlan** class that is used for the *SafeHome*
    - For the security functions, **FloorPlan** is used only during configuration activities and uses the operations *placeDevice(), showDevice(), groupDevice(),* and *removeDevice()* to place, show, group, and remove sensors from the floor plan.
    - The *SafeHome* surveillance function uses the four operations noted for security, but also requires special operations to manage cameras: *showFOV()* and *showDeviceID().*
    - Hence, the ISP suggests that client components from the two *SafeHome* functions have specialized interfaces defined for them.
    - The interface for security would encompass only the operations *placeDevice(), showDevice(), groupDevice(),* and *removeDevice().*
    - The interface for surveillance would incorporate the operations *placeDevice(),*

The Release Reuse Equivalency Principle (REP)

- *“The granule of reuse is the granule of release”*
- When classes or components are designed for reuse, there is an implicit contract that is established between the developer of the reusable entity and the people who will use it.
- The developer commits to establish a release control system that supports and maintains older versions of the entity while the users slowly upgrade to the most current version.
- Rather than addressing each class individually, it is often advisable to group reusable classes into packages that can be managed and controlled as newer versions evolve.

The Common Closure Principle (CCP)

- *“Classes that change together belong together”*
- Classes should be packaged cohesively.
- That is, when classes are packaged as part of a design, they should address the same functional or behavioral area.

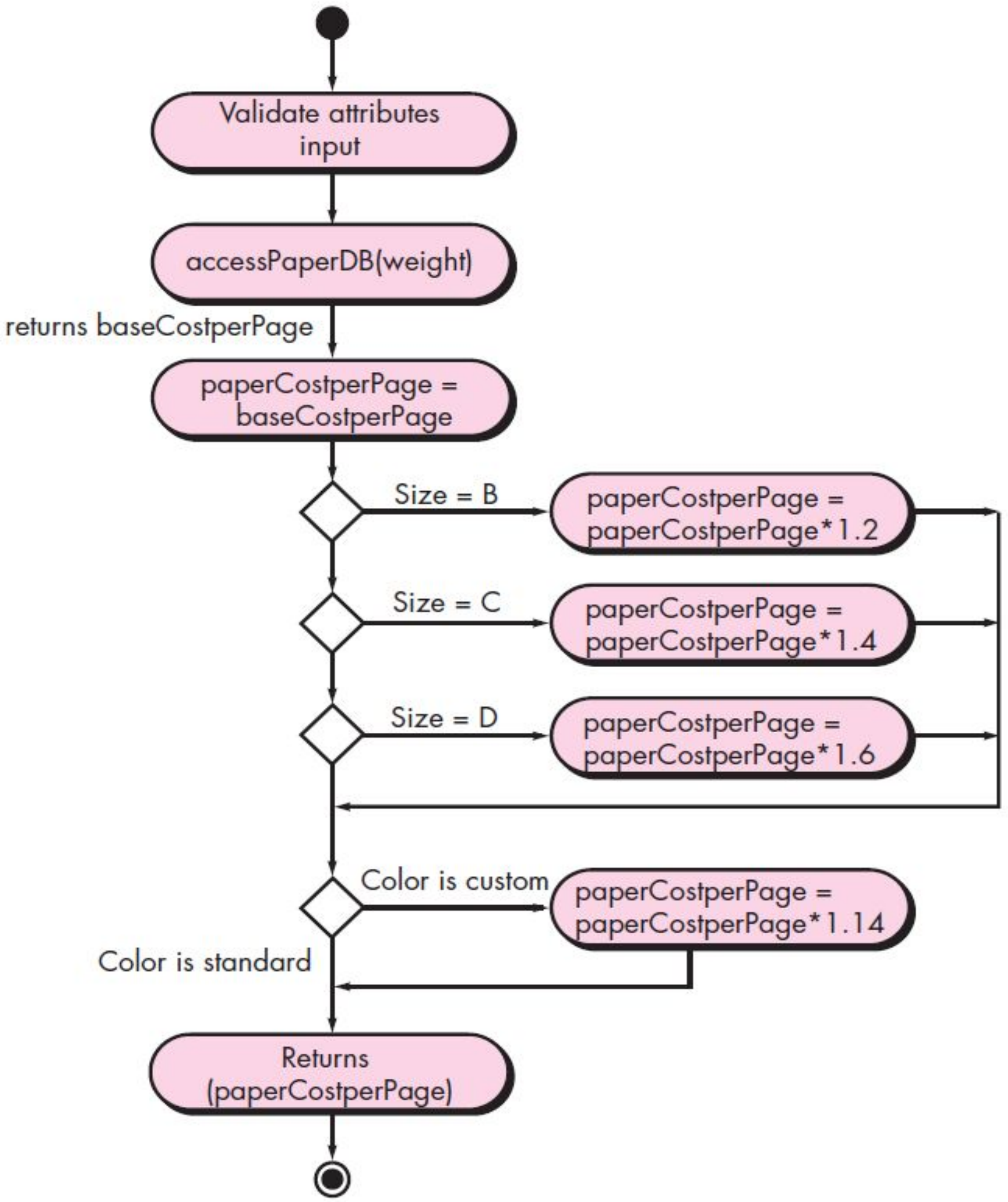
The Common Reuse Principle (CRP)

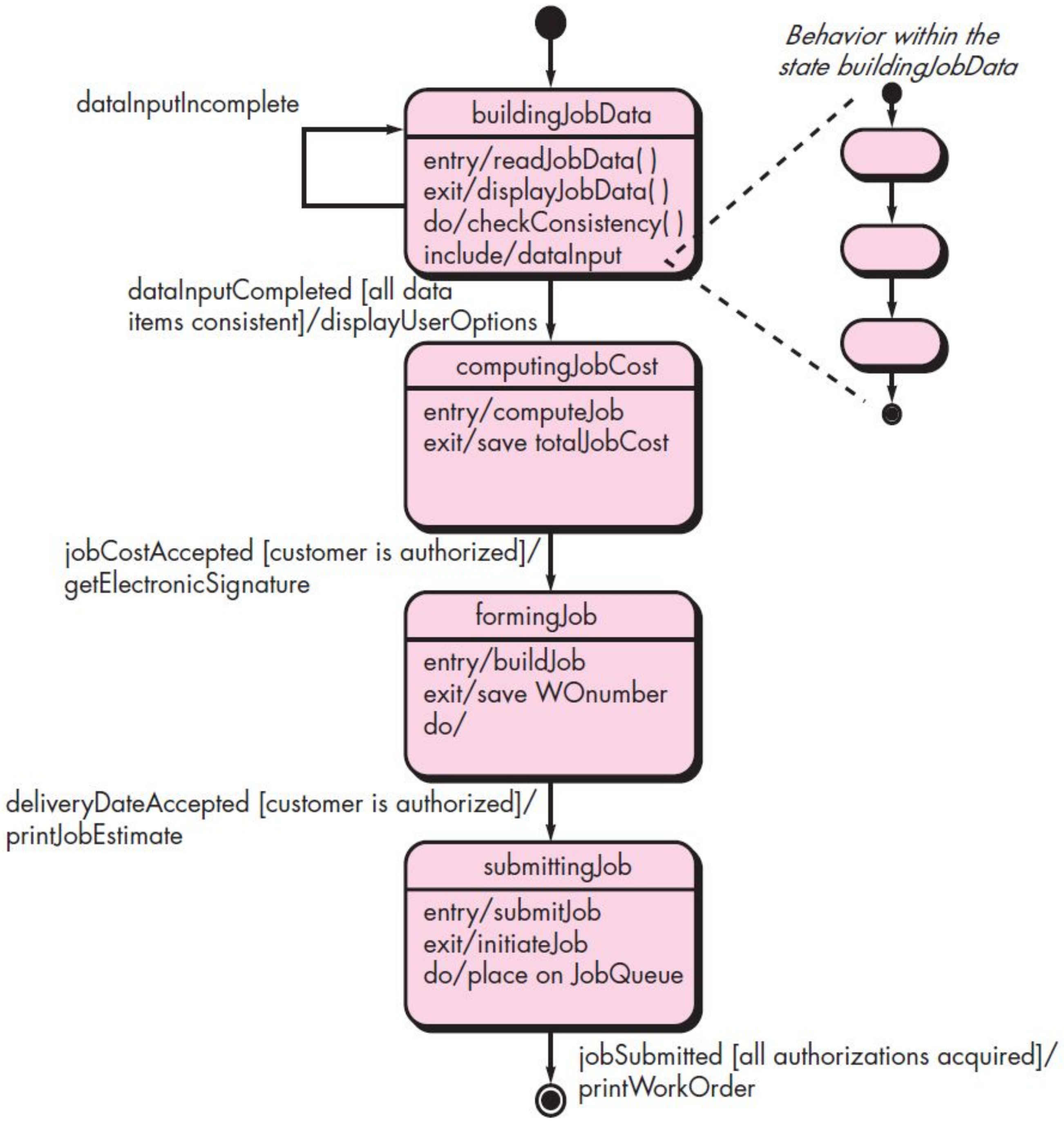
- *“Classes that aren’t reused together should not be grouped together”*
- This will precipitate unnecessary integration and testing.

- Cohesion and coupling basically used in Component Level Design.

Component-Leve Design

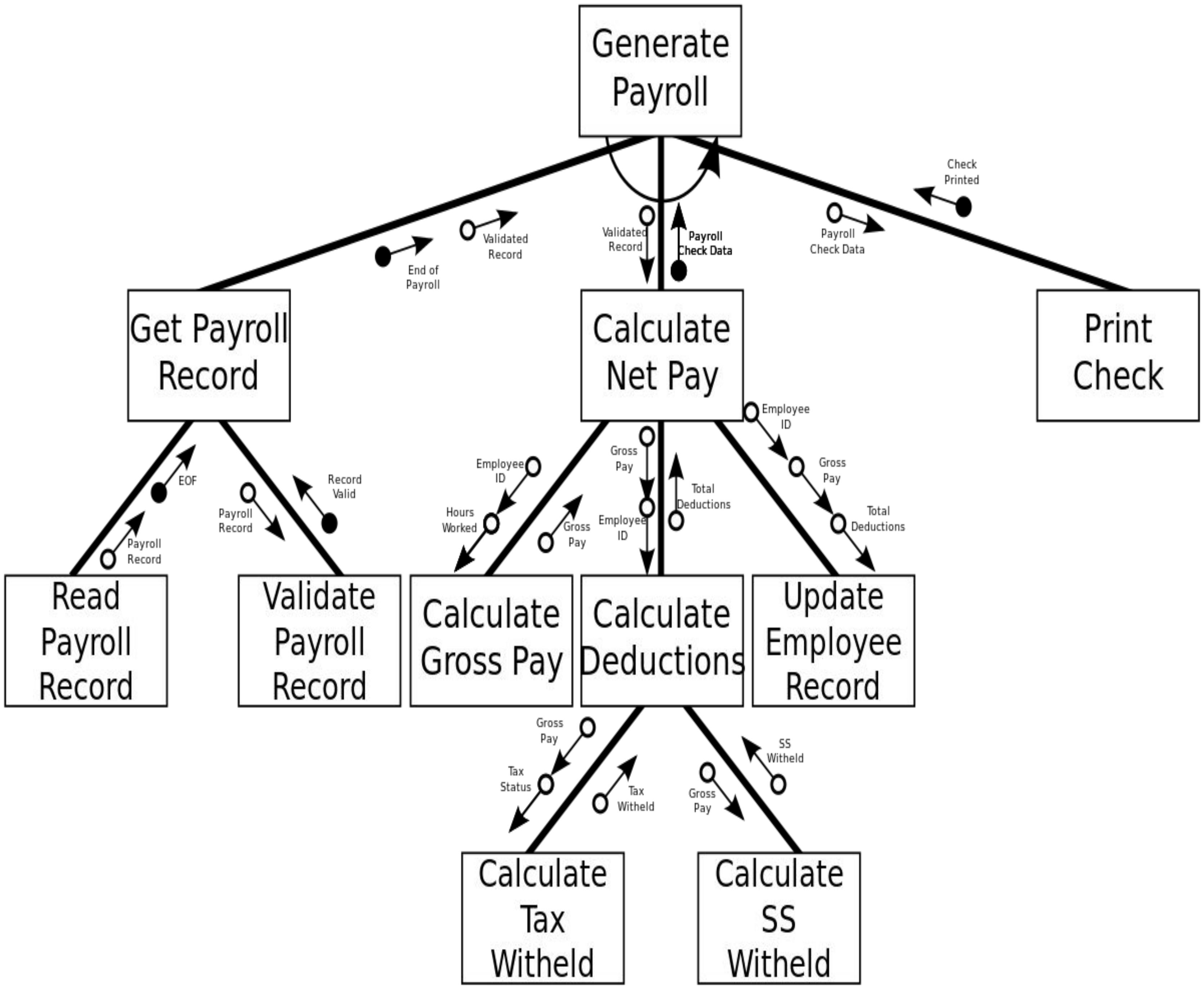
- Step 1. Identify all design classes that correspond to the problem domain.
- Step 2. Identify all design classes that correspond to the infrastructure domain.
- Step 3. Elaborate all design classes that are not acquired as reusable components.
  - Step 3a. Specify message details when classes or components collaborate.
  - Step 3b. Identify appropriate interfaces for each component.
  - Step 3c. Elaborate attributes and define data types and data structures required to implement them.
  - Step 3d. Describe processing flow within each operation in detail. (Activity diagram)
- Step 4. Describe persistent data sources (databases and files) and identify the classes required to manage them.
- Step 5. Develop and elaborate behavioral representations for a class or component. (statechart diagram)
- Step 6. Elaborate deployment diagrams to provide additional implementation detail.
- Step 7. Refactor every component-level design representation and always consider alternatives.


Validate attributes input
accessPaperDB(weight)
returns baseCostperPage
paperCostperPage = baseCostperPage
Size = B
paperCostperPage = paperCostperPage*1.2
Size = C
paperCostperPage = paperCostperPage*1.4
Size = D
paperCostperPage = paperCostperPage*1.6
Color is custom
paperCostperPage = paperCostperPage*1.14
Color is standard
Returns (paperCostperPage)


Behavior within the state buildingJobData
dataInputIncomplete
buildingJobData
entry/readJobData( )
exit/displayJobData( )
do/checkConsistency( )
include/dataInput
dataInputCompleted [all data items consistent]/displayUserOptions
computingJobCost
entry/computeJob
exit/save totalJobCost
jobCostAccepted [customer is authorized]/ getElectronicSignature
formingJob
entry/buildJob
exit/save WOnumber
do/
deliveryDateAccepted [customer is authorized]/ printJobEstimate
submittingJob
entry/submitJob
exit/initiateJob
do/place on JobQueue
jobSubmitted [all authorizations acquired]/ printWorkOrder

Structure Chart

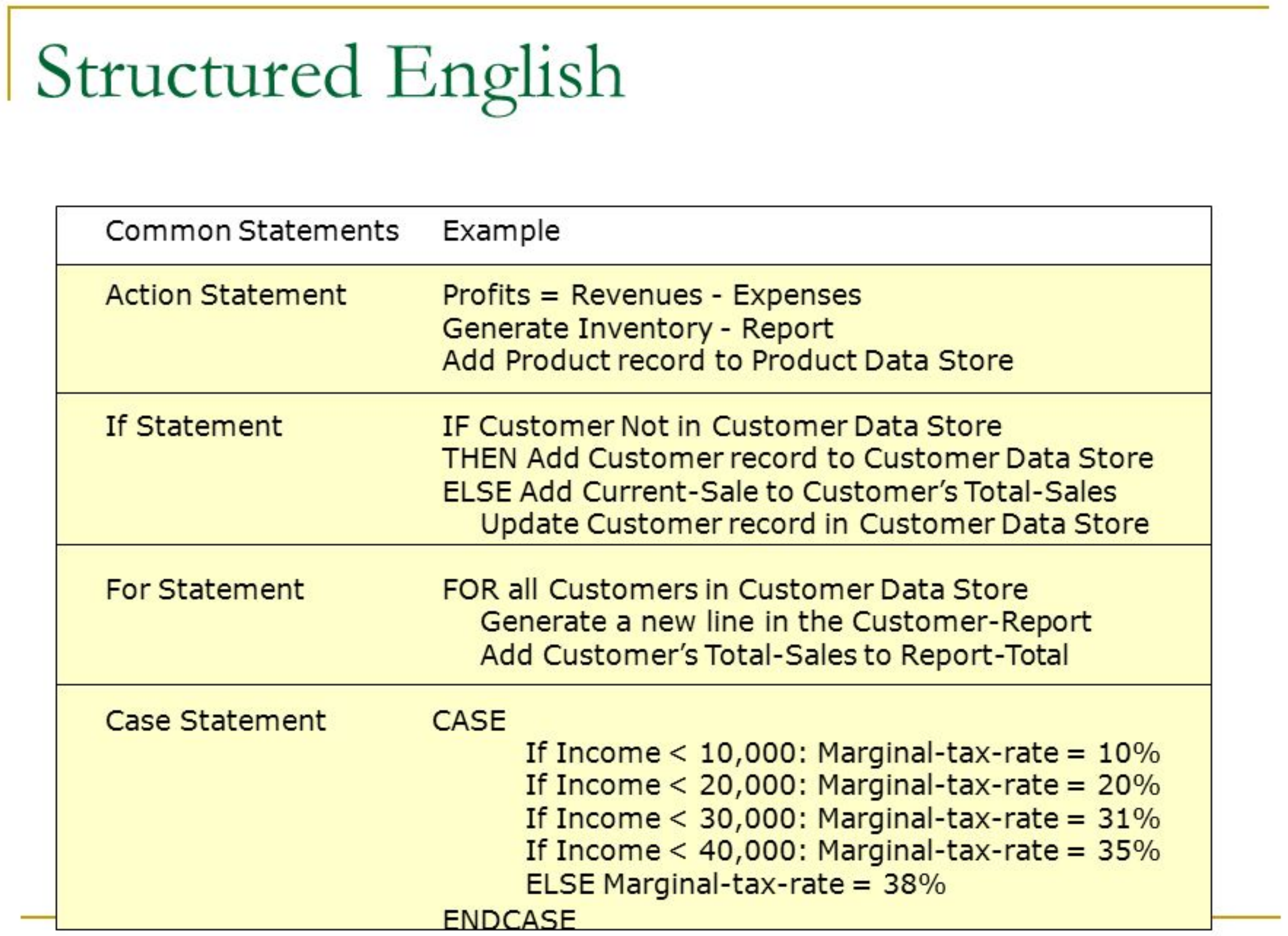
- **Structure Chart** represent hierarchical structure of modules.
- It breaks down the entire system into lowest functional modules, describe functions and sub-functions of each module of a system to a greater detail.



Structured English

- **Structured English** is the use of the English language with the syntax of structured programming to communicate the design of a computer program to non-technical users.
- It is done by breaking it down into logical steps using straightforward English words.
- Structured English gives aims to get the benefits of both the programming logic and natural language:
    - Program logic helps to attain precision,
    - Whilst natural language helps with the familiarity of the spoken word.
- It is the basis of some programming languages such as SQL.
- **Advanced English Structure** is a limited-form "pseudocode" and consists of the following elements:
    1. Operation statements written as English phrases executed from the top down
    2. Conditional blocks indicated by keywords such as IF, THEN, and ELSE
    3. Repetition blocks indicated by keywords such as DO, WHILE, and UNTIL

Pseudocode



Structured English

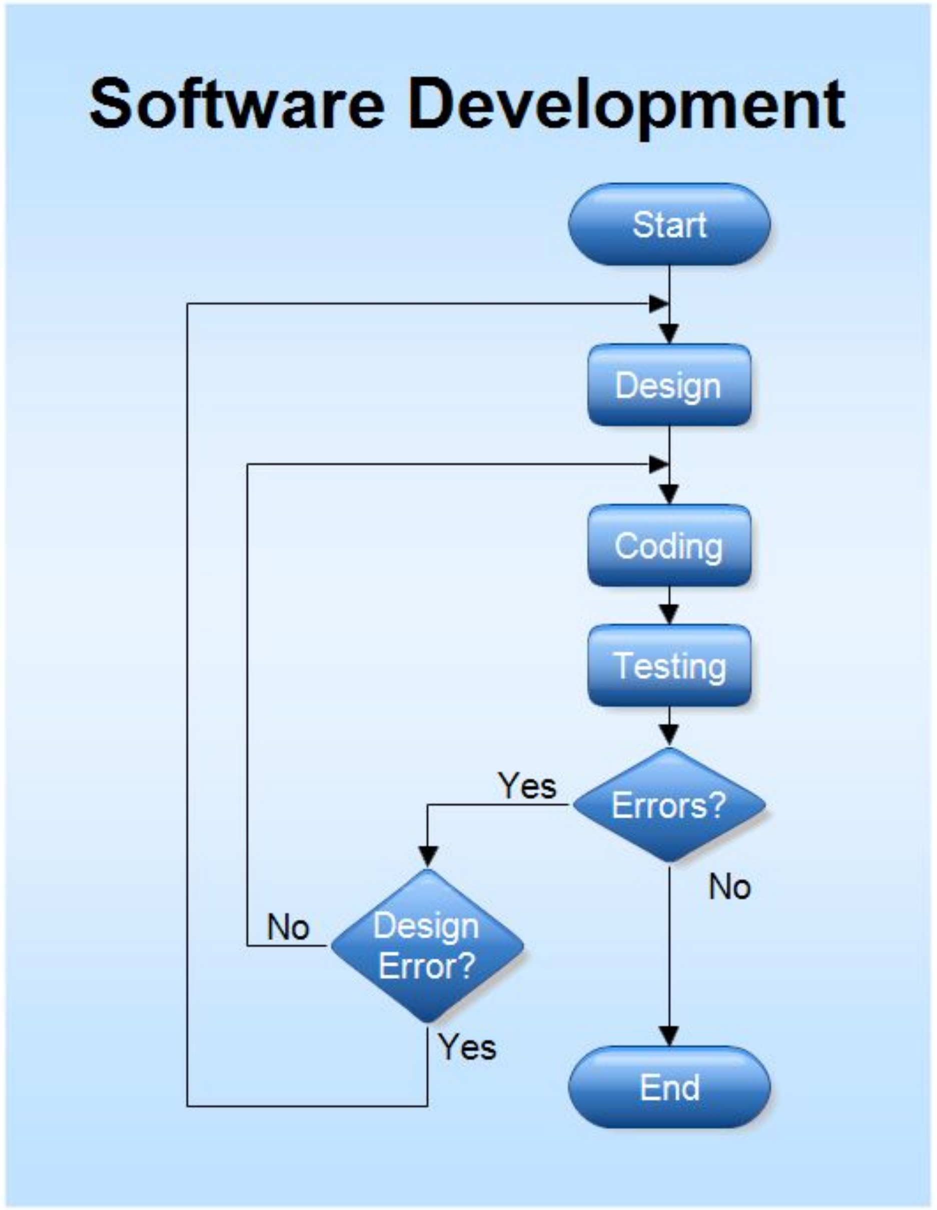
Common Statements	Example
Action Statement	Profits = Revenues - Expenses Generate Inventory - Report Add Product record to Product Data Store

If Statement	IF Customer Not in Customer Data Store
	THEN Add Customer record to Customer Data Store
	ELSE Add Current-Sale to Customer's Total-Sales
	    Update Customer record in Customer Data Store

For Statement	FOR all Customers in Customer Data Store Generate a new line in the Customer-Report Add Customer's Total-Sales to Report-Total

Case Statement	CASE
	        If Income < 10,000: Marginal-tax-rate = 10%
	        If Income < 20,000: Marginal-tax-rate = 20%
	        If Income < 30,000: Marginal-tax-rate = 31%
	        If Income < 40,000: Marginal-tax-rate = 35%
	        ELSE Marginal-tax-rate = 38%
	ENDCASE

Flowchart


Software Development
Start
Design
Coding
Testing
Errors?
Yes
No
Design Error?
No
Yes
End

Decision Table

- *Use a decision table when a complex set of conditions and actions are encountered within a component.*
- *Decision tables* provide a notation that translates actions and conditions into a tabular form.
- 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 that 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.*

Conditions	1	2	3	4	5	6
Regular customer	T	T				
Silver customer			T	T		
Gold customer					T	T
Special discount	F	T	F	T	F	T
Actions						
No discount	✓					
Apply 8 percent discount			✓	✓		
Apply 15 percent discount					✓	✓
Apply additional *x* percent discount		✓		✓		✓

To illustrate the use of a decision table, consider the following excerpt from an informal use case that has just been proposed for the print shop system:

> Three types of customers are defined: a regular customer, a silver customer, and a gold customer (these types are assigned by the amount of business the customer does with the print shop over a 12 month period). A regular customer receives normal print rates and delivery. A silver customer gets an 8 percent discount on all quotes and is placed ahead of all regular customers in the job queue. A gold customer gets a 15 percent reduction in quoted prices and is placed ahead of both regular and silver customers in the job queue. A special discount of *x* percent in addition to other discounts can be applied to any customer's quote at the discretion of management.