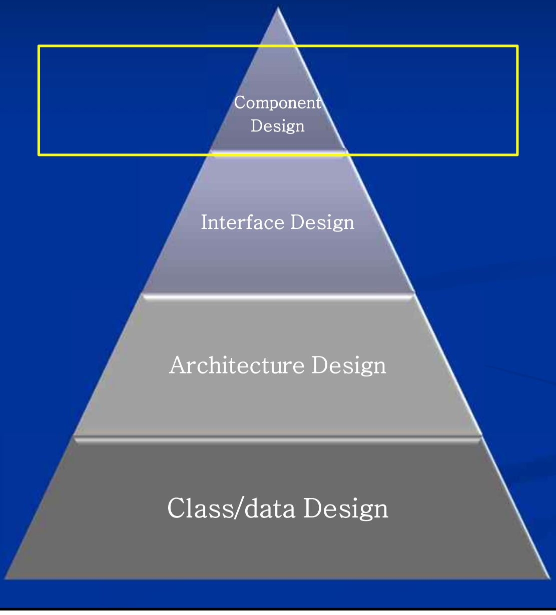
컴포넌트 설계

한국항공대학교 소프트웨어학과 지승도교수
R.S. Pressman

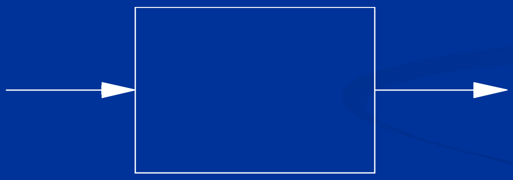
컴포넌트 설계



Monolithic vs. Modular

- Unified

➔ constructed by one single module

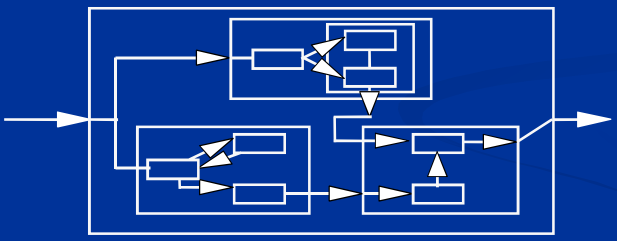


Monolithic

Monolithic vs. Modular

- Hierarchical, Abstraction, Divide & conquer

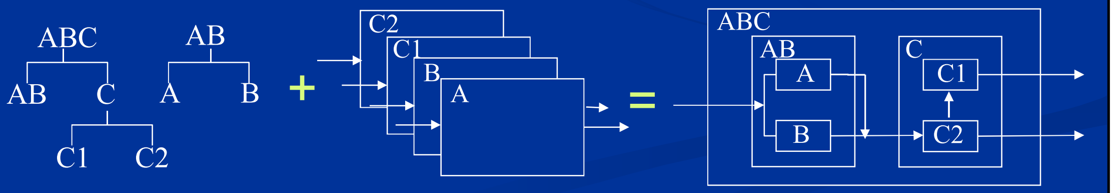
➔ constructed by multiple module in hierarchical fashion



Multi-facetted

컴포넌트란?

- *Component is a modular building block for software*
- "a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces." (UML view-point)



컴포넌트란? (계속)

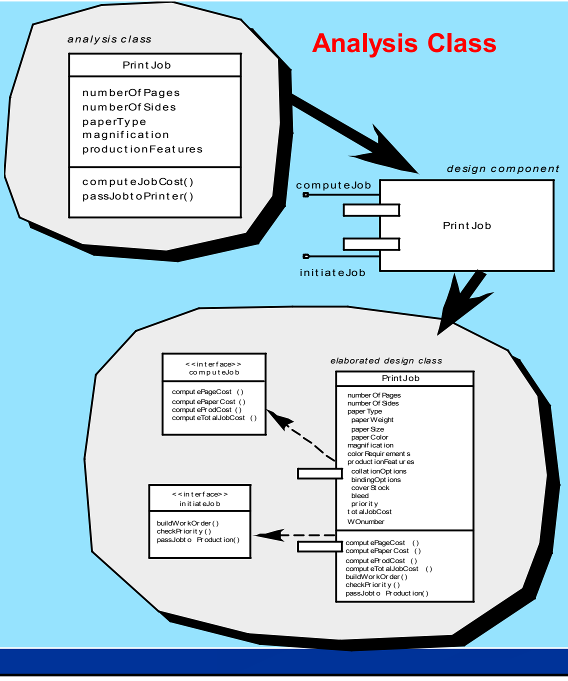
- OOD view (Object-Oriented Design):

➔ a component contains a set of collaborating classes. Each class within a component has been fully elaborated to include all attributes and operations that are relevant to its implementation.

- Conventional view (Structured Design):

➔ processing logic, the internal data structures that are required to implement the processing logic, and an interface that enables the component to be invoked and data to be passed to it.

Object-oriented Component Design

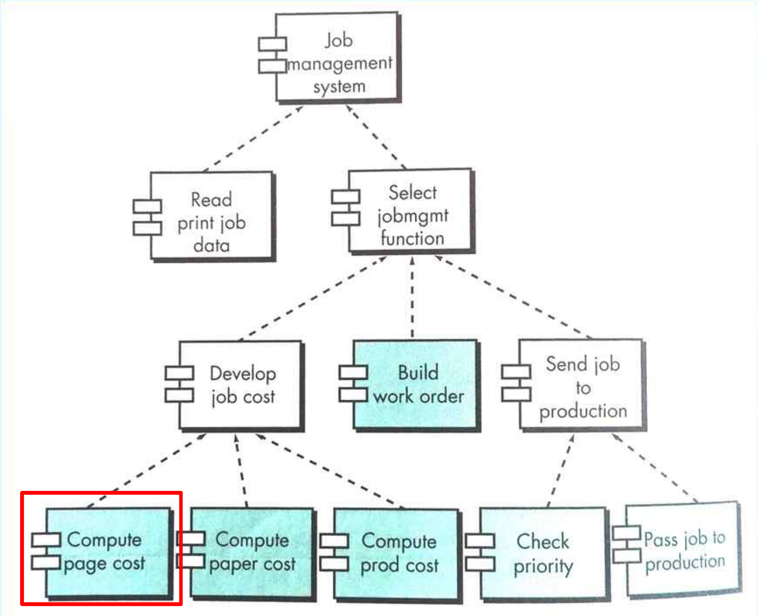


Analysis Class

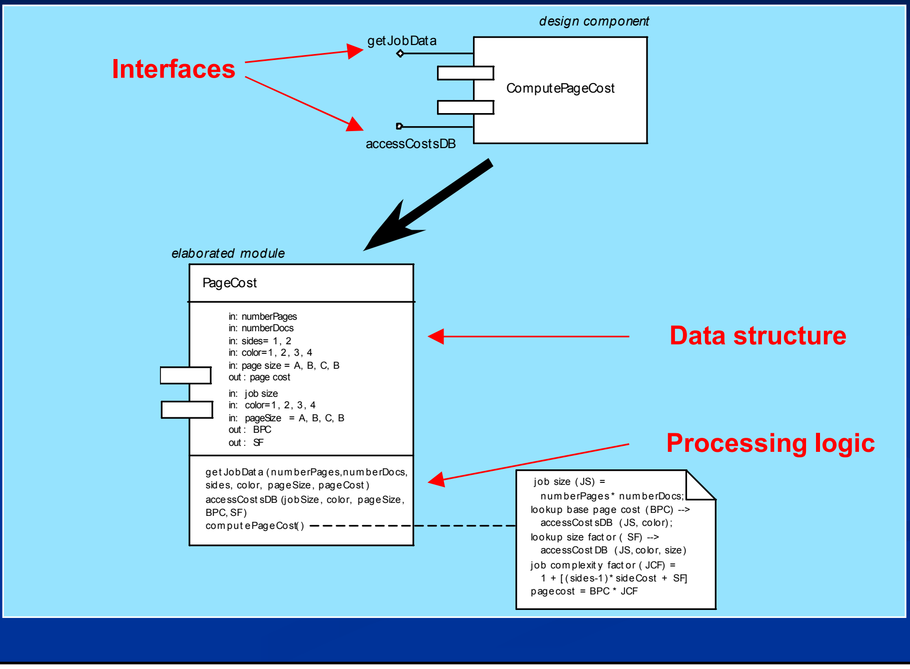
Component
(in architecture design)

Design Class
(in component design)
- **Collaborating classes**
- **Attributes**
- **Operations**

Structured Component Design



Structured Component Design



```
in: numberPages
in: numberDocs
in: sides= 1, 2
in: color=1, 2, 3, 4
in: page size = A, B, C, B
out : page cost
in:  job size
in:  color=1, 2, 3, 4
in:  pageSize  = A, B, C, B
out :  BPC
out :  SF
```

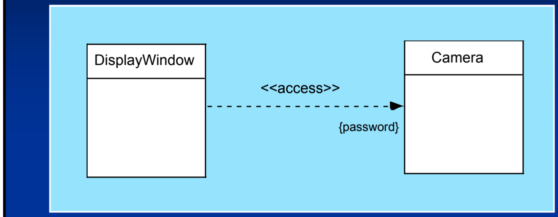
```
getJobData (numberPages,numberDocs,
sides, color, pageSize, pageCost)
accessCostsDB (jobSize, color, pageSize,
BPC, SF)
computePageCost( )
```

```
 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
```

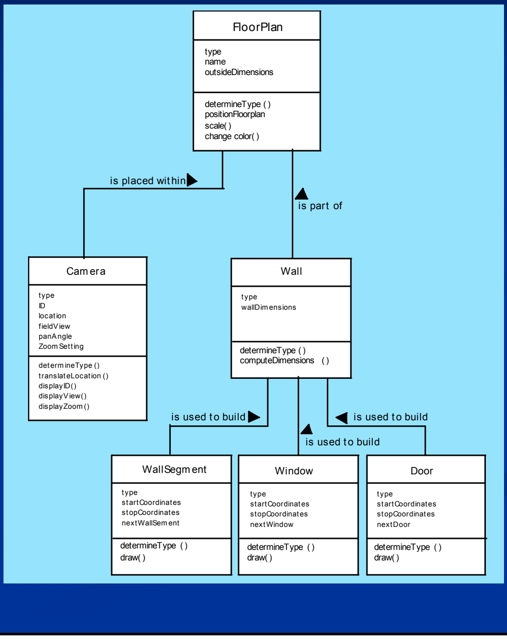
컴포넌트 설계 지침

- Components
  - Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model (problem-oriented → implementation-specific)
- Interfaces
  - Interfaces provide important information about communication and collaboration
- Dependencies and Inheritance
  - it is a good idea to model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes).

Dependency



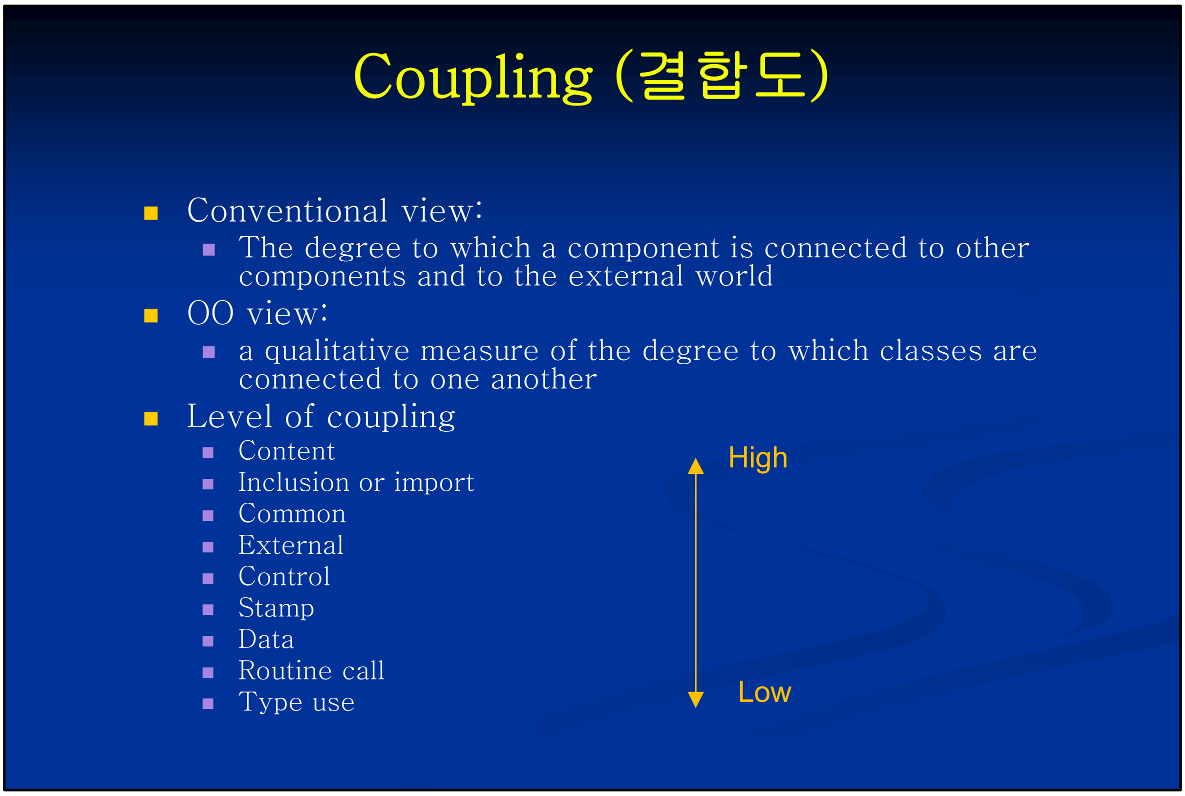
Inheritance



Cohesion (응집도)

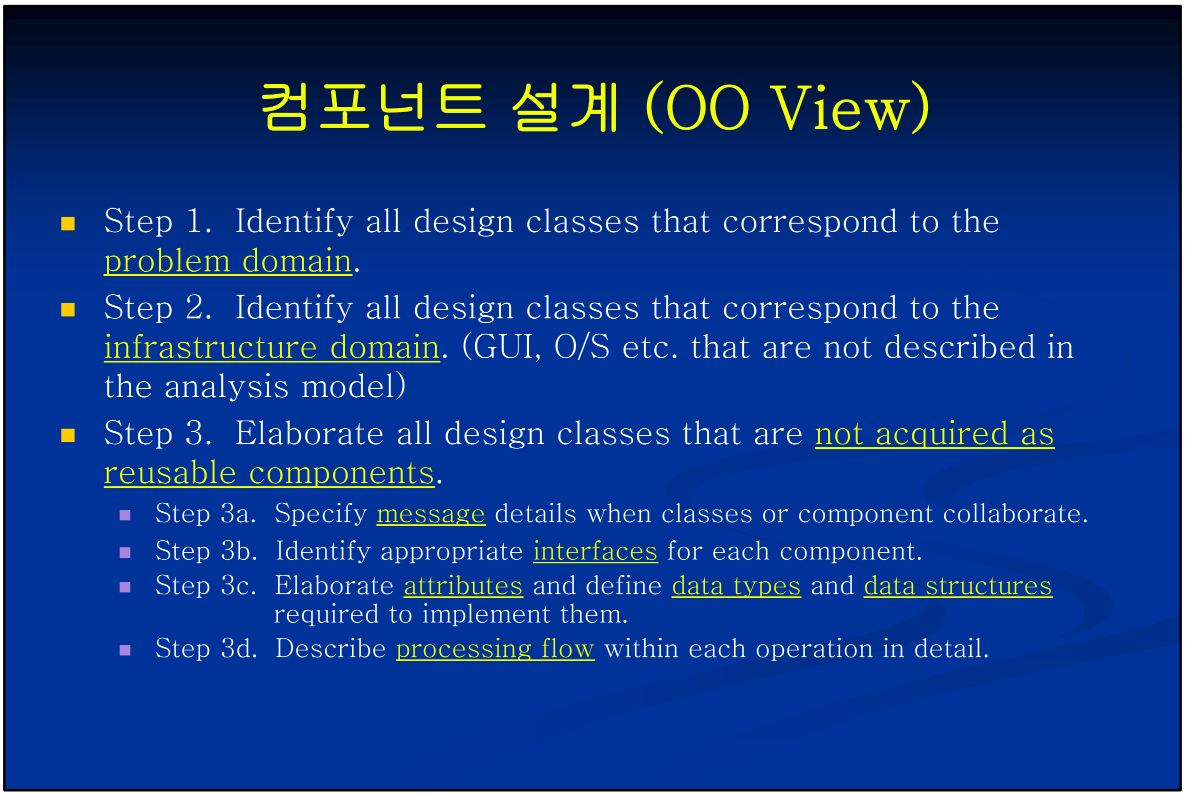
- Conventional view:
  - "single-mindedness" of a module
- OO view:
  - *cohesion* implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself
- Levels of cohesion
  - Functional
  - Layer
  - Communicational
  - Sequential
  - Procedural
  - Temporal
  - Utility

High ↕ Low



Coupling (결합도)

- Conventional view:
  - The degree to which a component is connected to other components and to the external world
- OO view:
  - a qualitative measure of the degree to which classes are connected to one another
- Level of coupling
  - Content
  - Inclusion or import
  - Common
  - External
  - Control
  - Stamp
  - Data
  - Routine call
  - Type use



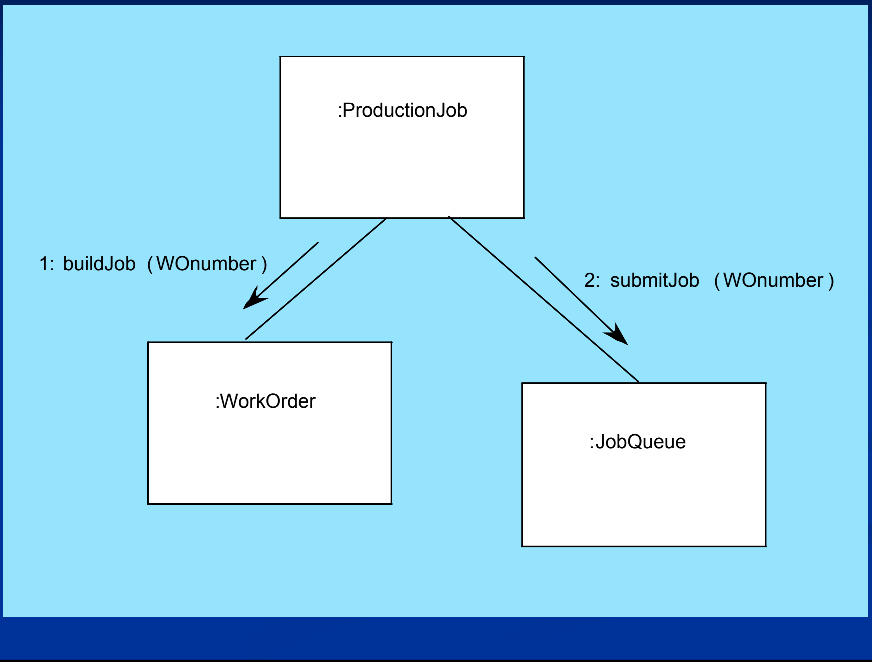
컴포넌트 설계 (OO View)

- Step 1. Identify all design classes that correspond to the problem domain.
- Step 2. Identify all design classes that correspond to the infrastructure domain. (GUI, O/S etc. that are not described in the analysis model)
- Step 3. Elaborate all design classes that are not acquired as reusable components.
  - Step 3a. Specify message details when classes or component 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.

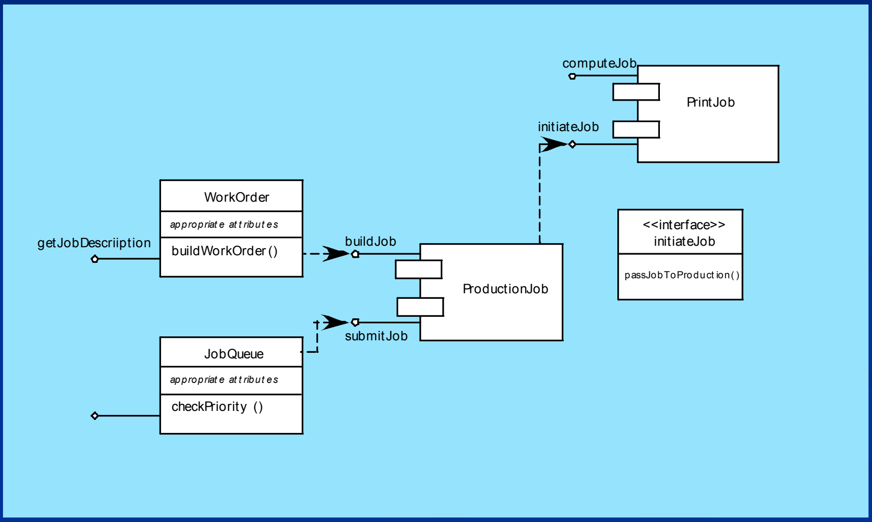
컴포넌트 설계 (계속)

- 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.
- Step 6. Elaborate deployment diagrams to provide additional implementation detail.
- Step 7. Factor every component-level design representation and always consider alternatives.

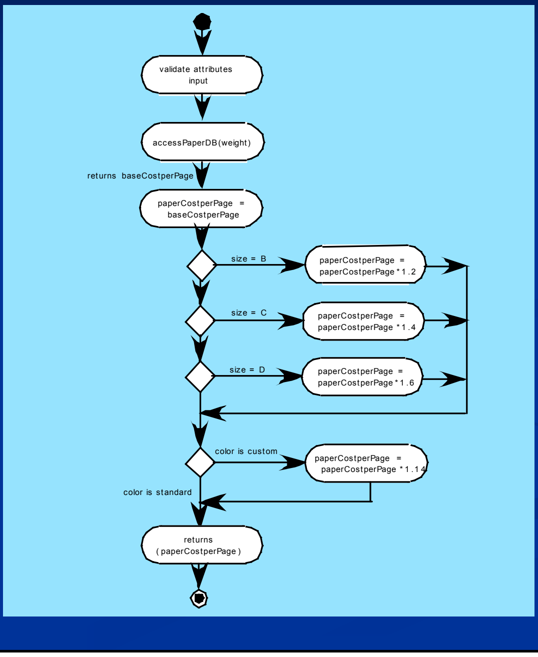
Collaboration Diagram



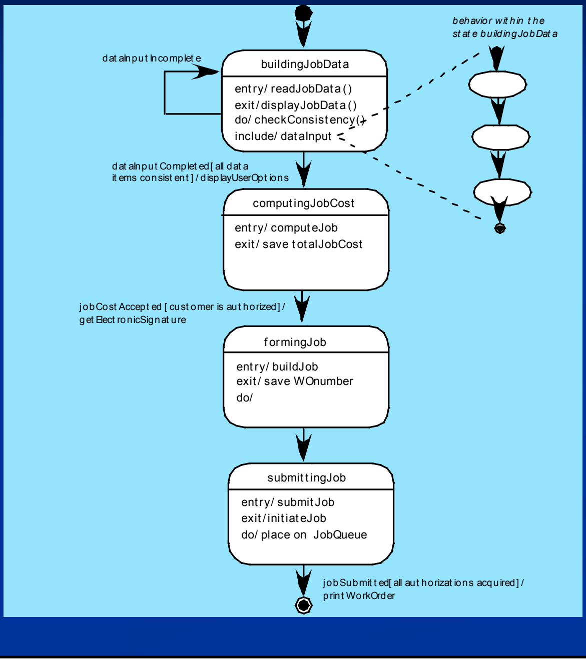
Refactoring Interface



Activity Diagram


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)

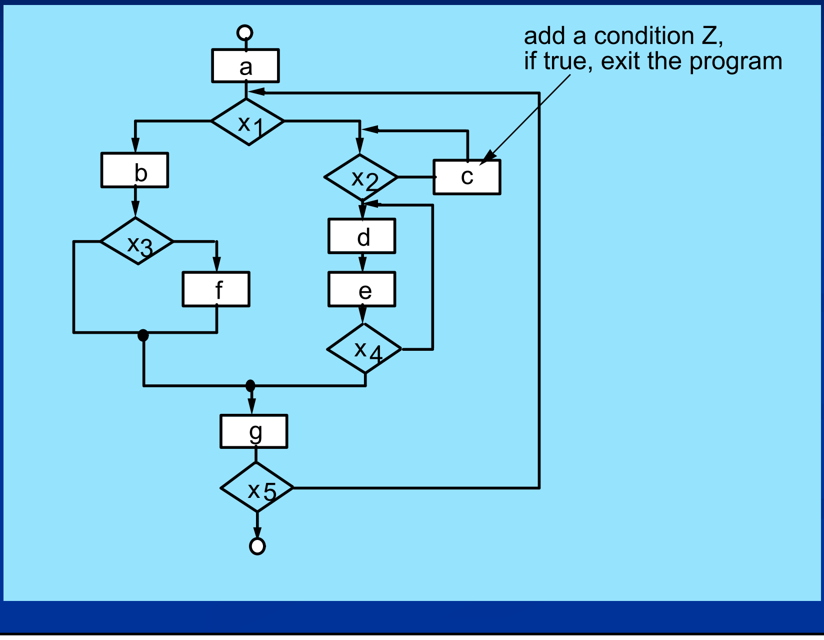
Statechart Diagram



Designing Conventional Components

- uses a limited set of logical constructs: “Structured Programming”
  - *sequence*
  - *condition* if-then-else, case
  - *loops*
- options:
  - graphical (e.g. flowchart, box diagram)
  - decision table
  - Pseudo-code (e.g., PDL: Program Design Language)

Flow Chart

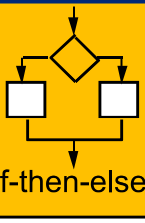


add a condition Z,
if true, exit the program

Decision Table

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
Rules						
no discount	✓					
apply 8 percent discount			✓	✓		
apply 15 percent discount					✓	✓
apply additional x percent discount		✓		✓		✓

Program Design Language (PDL)



if-then-else

```
if condition x
  then process a;
  else process b;
endif
```

PDL

- □ easy to combine with source code
- □ machine readable, no need for graphics input
- □ graphics can be generated from PDL
- □ enables declaration of data as well as procedure
- □ easier to maintain