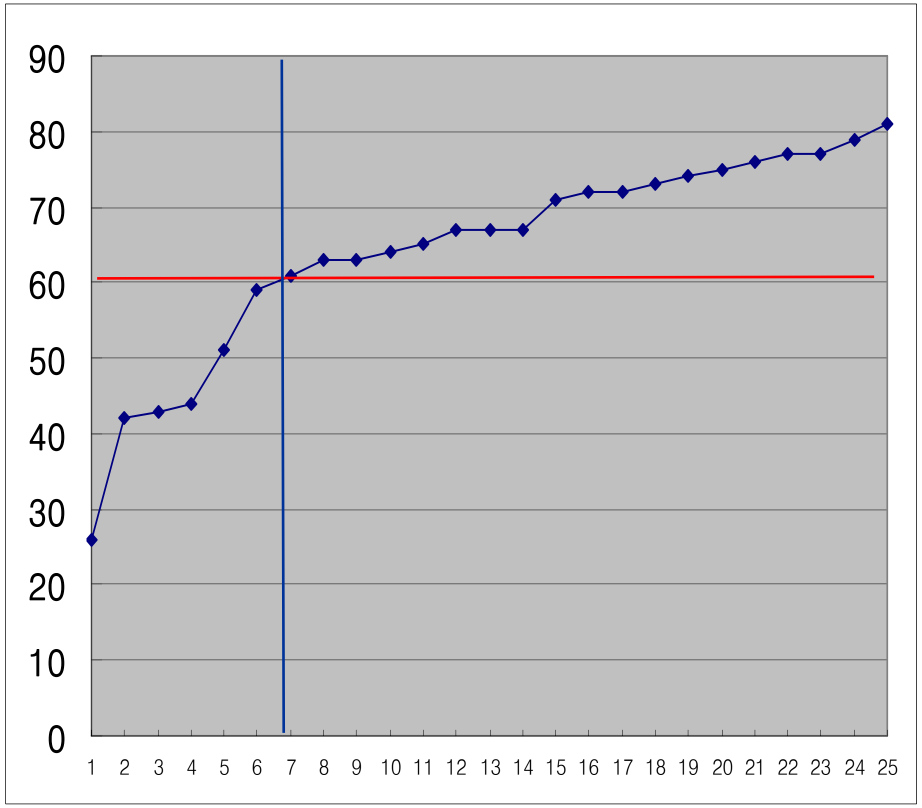
Midterm Exam Statistics

- Statistics
  - Average: 60.9
  - Highest score :81
  - Lowest score: 26
- Linear distribution of scores
  - Your efforts of studying pay you back well
  - Spend time and energy to read SEPA !!!



Chapter 11
Component-Level Design

Moonzoo Kim

CS Division of EECS Dept.
KAIST
moonzoo@cs.kaist.ac.kr
http://pswlab.kaist.ac.kr/courses/cs550-07

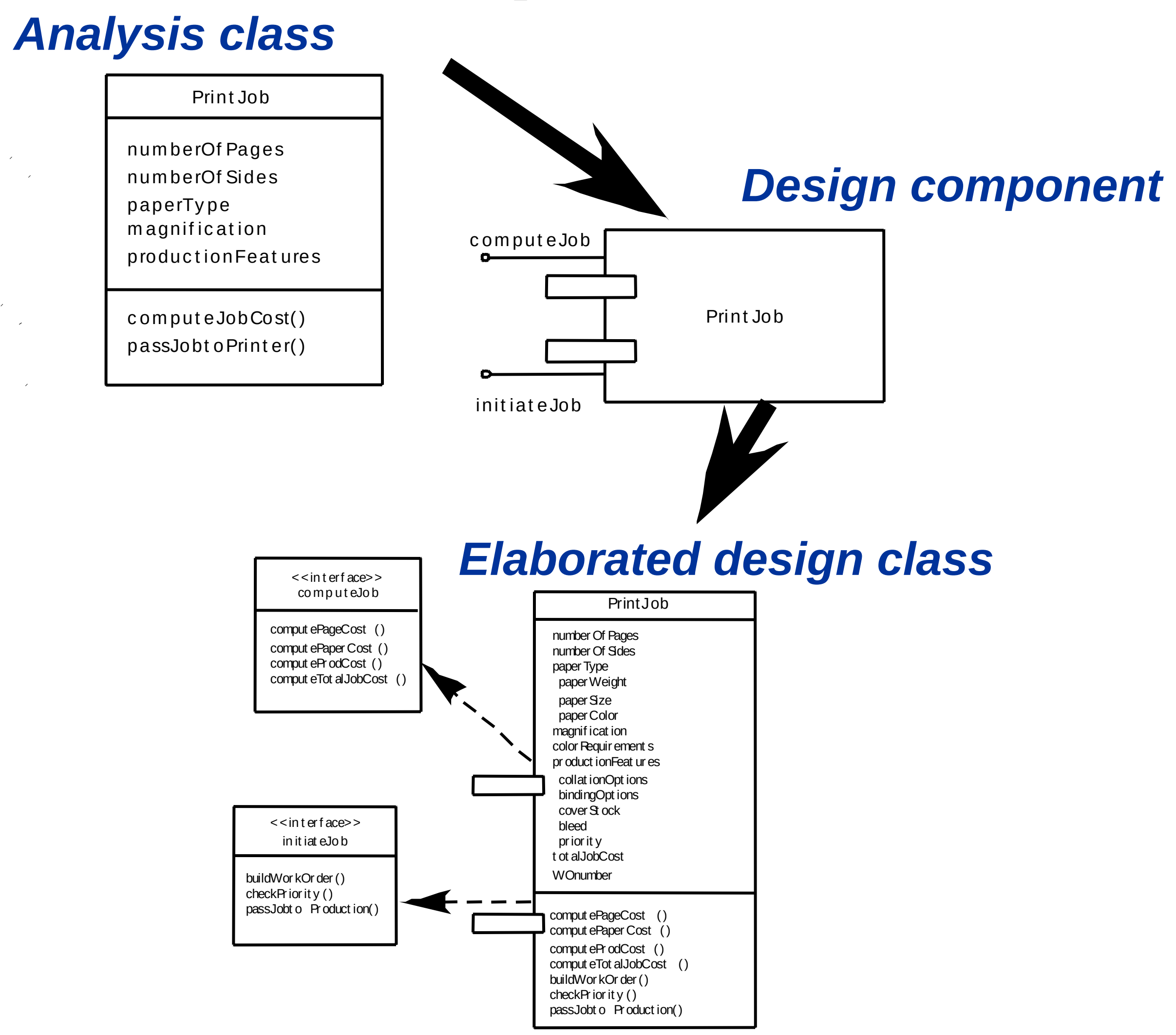
Overview of Ch 11. modeling Component-level Design

- 11.1 What is a component
  - An object-oriented view
- 11.2 Designing class-based components
  - Basic design principles
  - Component-level design guidelines
  - Cohesion
  - Coupling
- 11.3 Conducting component-level design
- 11.4 Object constraint language (OCL)
- 11.5 Designing conventional components
  - Graphical design notation
  - Tabular design notation
  - Program design language

What is a Component?

- *OMG Unified Modeling Language Specification* [OMG01] defines a component as
  - "… a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."
- OO view: a component contains a set of collaborating classes
- Conventional view: 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.

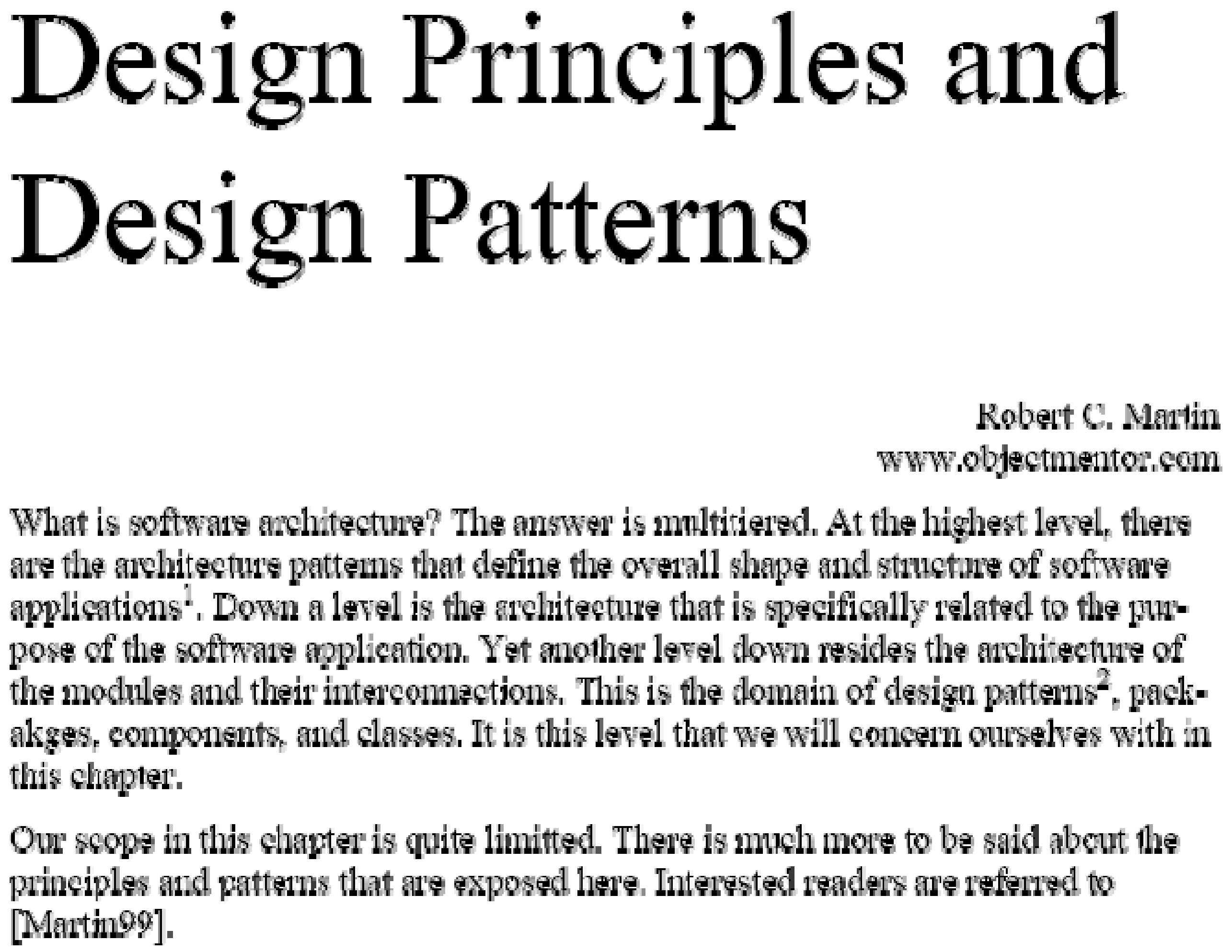
OO Component



Basic Design Principles

- The Open-Closed Principle (OCP).
  - *"A module [component] should be open for extension but closed for modification.*
- The Liskov Substitution Principle (LSP).
  - *"Subclasses should be substitutable for their base classes.*
- Dependency Inversion Principle (DIP).
  - *"Depend on abstractions. Do not depend on concretions."*
- The Interface Segregation Principle (ISP).
  - *"Many client-specific interfaces are better than one general purpose interface.*
- The Release Reuse Equivalency Principle (REP).
  - *"The granule of reuse is the granule of release."*
- The Common Closure Principle (CCP).
  - *"Classes that change together belong together."*
- The Common Reuse Principle (CRP).
  - *"Classes that aren't reused together should not be grouped together."*

Source: Martin, R., "Design Principles and Design Patterns," downloaded from http://www.objectmentor.com, 2000.



Design Principles and Design Patterns

Robert C. Martin
www.objectmentor.com

What is software architecture? The answer is multitiered. At the highest level, there are the architecture patterns that define the overall shape and structure of software applications[1]. Down a level is the architecture that is specifically related to the purpose of the software application. Yet another level down resides the architecture of the modules and their interconnections. This is the domain of design patterns[2], packakges, components, and classes. It is this level that we will concern ourselves with in this chapter.

Our scope in this chapter is quite limitted. There is much more to be said about the principles and patterns that are exposed here. Interested readers are referred to [Martin99].

The OCP in Action (pg332)

- **The scene:**
  - Vinod's cubicle.
- **The players:**
  - Vinod, Shakira

    members of the *SafeHome* software engineering team.
- **The conversation:**
- **Vinod:** I just got a call from Doug [the team manager]. He says marketing wants to add a new sensor.
- **Shakira (smirking):** Not again, jeez!
- **Vinod:** Yeah ... and you're not going to believe what these guys have come up with.
- **Shakira:** Amaze me.
- **Vinod (laughing):** They call it a doggie angst sensor.
- **Shakira:** Say what?
- **Vinod:** It's for people who leave their pets home in apartments or condos or houses that are close to one another. The dog starts to bark. The neighbor gets angry and complains. With this sensor, if the dog barks for more than, say, a minute, the sensor sets a special alarm mode that calls the owner on his or her cell phone.
- **Shakira:** You're kidding me, right?

- **Vinod:** Nope. Doug wants to know how much time it's going to take to add it to the security function.
- **Shakira (thinking a moment):** Not much ... look. [She shows Vinod Figure 11.4] We've isolated the actual sensor classes behind the **sensor** interface. As long as we have specs for the doggie sensor, adding it should be a piece of cake. Only thing I'll have to do is create an
- appropriate component ... uh, class, for it. No change to the **Detector** component at all.
- **Vinod:** So I'll tell Doug it's no big deal.
- **Shakira:** Knowing Doug, he'll keep us focused and not deliver the doggie thing until the next release.
- **Vinod:** That's not a bad thing, but can you implement now if he wants you to?
- **Shakira:** Yeah, the way we designed the interface lets me do it with no hassle.

Design Guidelines

- 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
- Interfaces
  - Interfaces provide important information about communication and collaboration (as well as helping us to achieve the OPC)
- 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).

Cohesion

- 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

Cohesion in Action (pg336-337)

- **The scene:**
  - Jamie's cubicle.
- **The players:**
  - Jamie, Ed

    members of the *SafeHome* software engineering team who are working on the surveillance function.
- **The conversation:**
- Ed**:** I have a first-cut design of the camera component.
- Jamie**:** Wanna do a quick review?
- Ed**:** I guess ... but really, I'd like your input on something.
- (Jamie gestures for him to continue.)
- Ed**:** We originally defined five operations for **camera.** Look ... [shows Jamie the list]
  - *determineType()* tells me the type of camera.
  - *translateLocation()* allows me to move the camera around the floor plan.
  - *displayID()* gets the camera ID and displays it near the camera icon.
  - *displayView()* shows me the field of view of the camera graphically.
  - *displayZoom()* shows me the magnification of the camera graphically.
- Ed**:** I've designed each separately, and they're pretty simple operations. So I thought

it might be a good idea to combine all of the display operations into just one that's called displayCamera()--it'll show the ID, the view, and the zoom. Whaddaya think?

- **Jamie (grimacing):** Not sure that's such a good idea.
- **Ed (frowning):** Why? All of these little ops can cause headaches.
- **Jamie:** The problem with combining them is we lose cohesion. You know, the *displayCamera()* op won't be single-minded.
- **Ed (mildly exasperated):** So what? The whole thing will be less than 100 source lines, max. It'll be easier to implement, I think.
- **Jamie:** And what if marketing decides to change the way that we represent the view field?
- **Ed:** I'll just jump into the *displayCamera()* op and make the mod.
- **Jamie:** What about side effects?
- **Ed:** Whaddaya mean?
- **Jamie:** Well, say you make the change but inadvertently create a problem with the ID display.

- **Ed:** I wouldn't be that sloppy.
- **Jamie:** Maybe not, but what if some support person two years from now has to make the mod. He might not understand the op as well as you do and, who knows, he might be sloppy.
- **Ed:** So you're against it?
- **Jamie:** You're the designer . . . it's your decision . . . just be sure you understand the consequences of low cohesion.
- **Ed (thinking a moment):** Maybe we'll go with separate display ops.
- **Jamie:** Good decision.

Coupling

- 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
  - Common
  - Control
  - Stamp
  - Data
  - Routine call
  - Type use
  - Inclusion or import
  - External

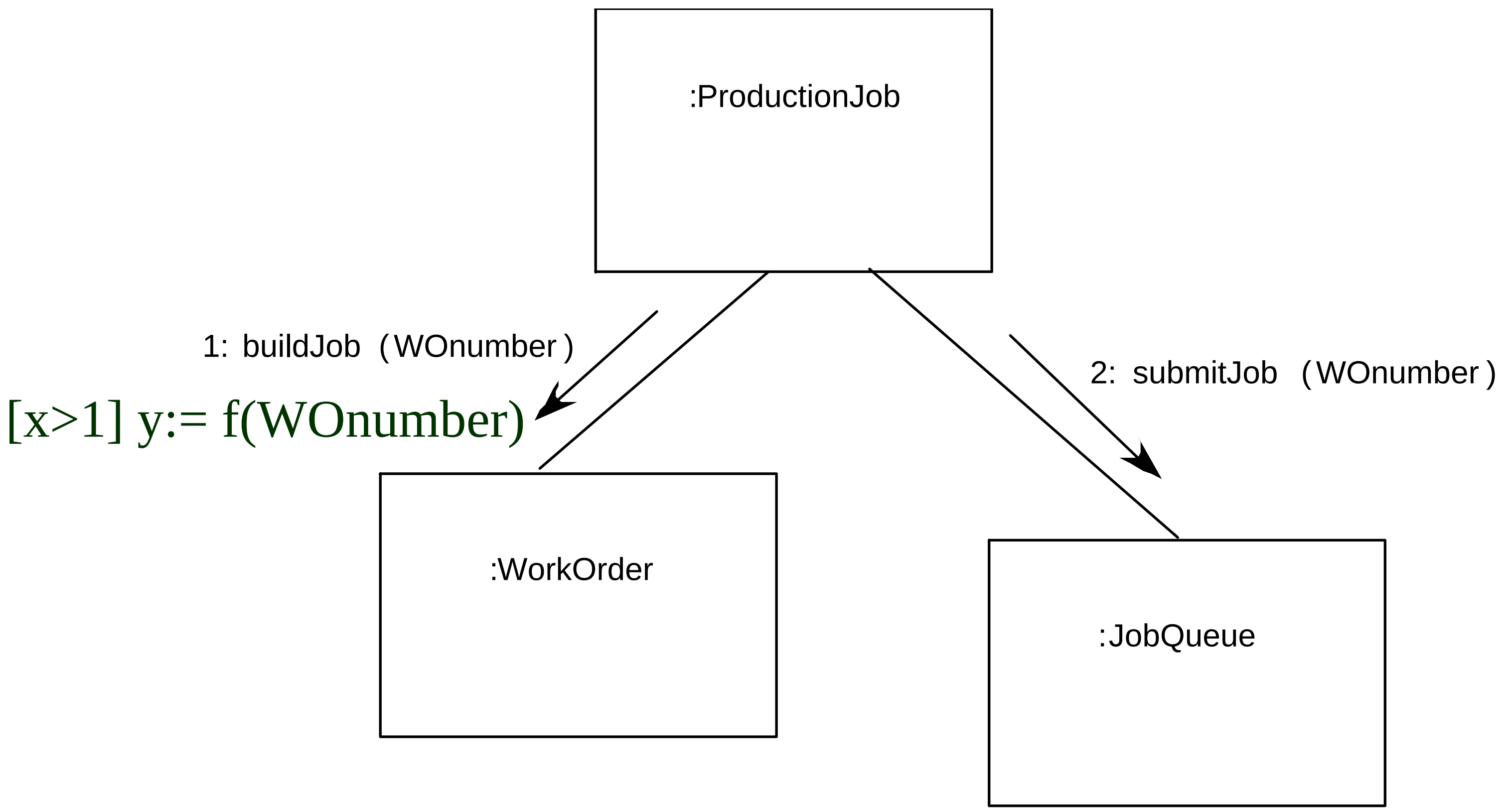
Component Level Design-I

- 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 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 (activity diagram) within each operation in detail.

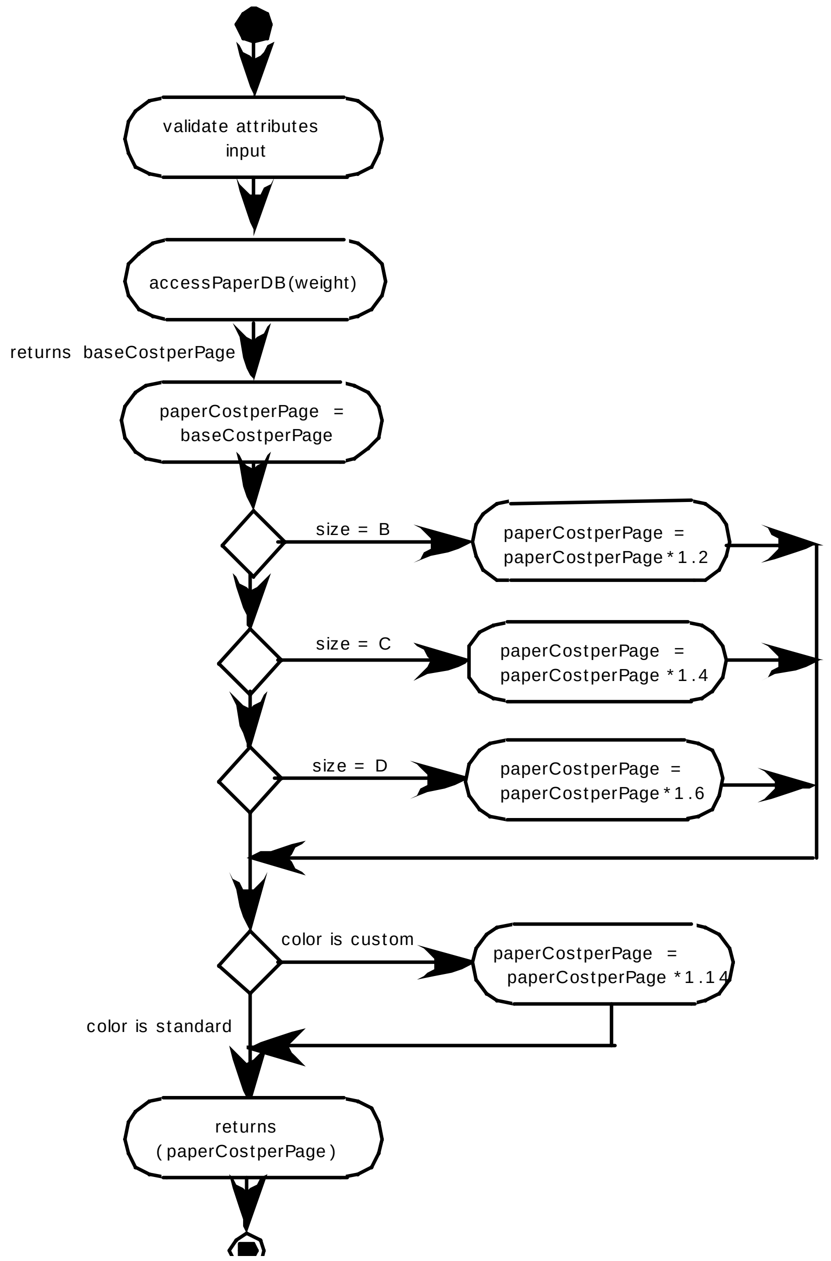
Component-Level Design-II

- Step 4. Describe persistent data sources (databases and files) and identify the classes required to manage them.
- Step 5. Develop and elaborate behavioral representations (statechart) 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.

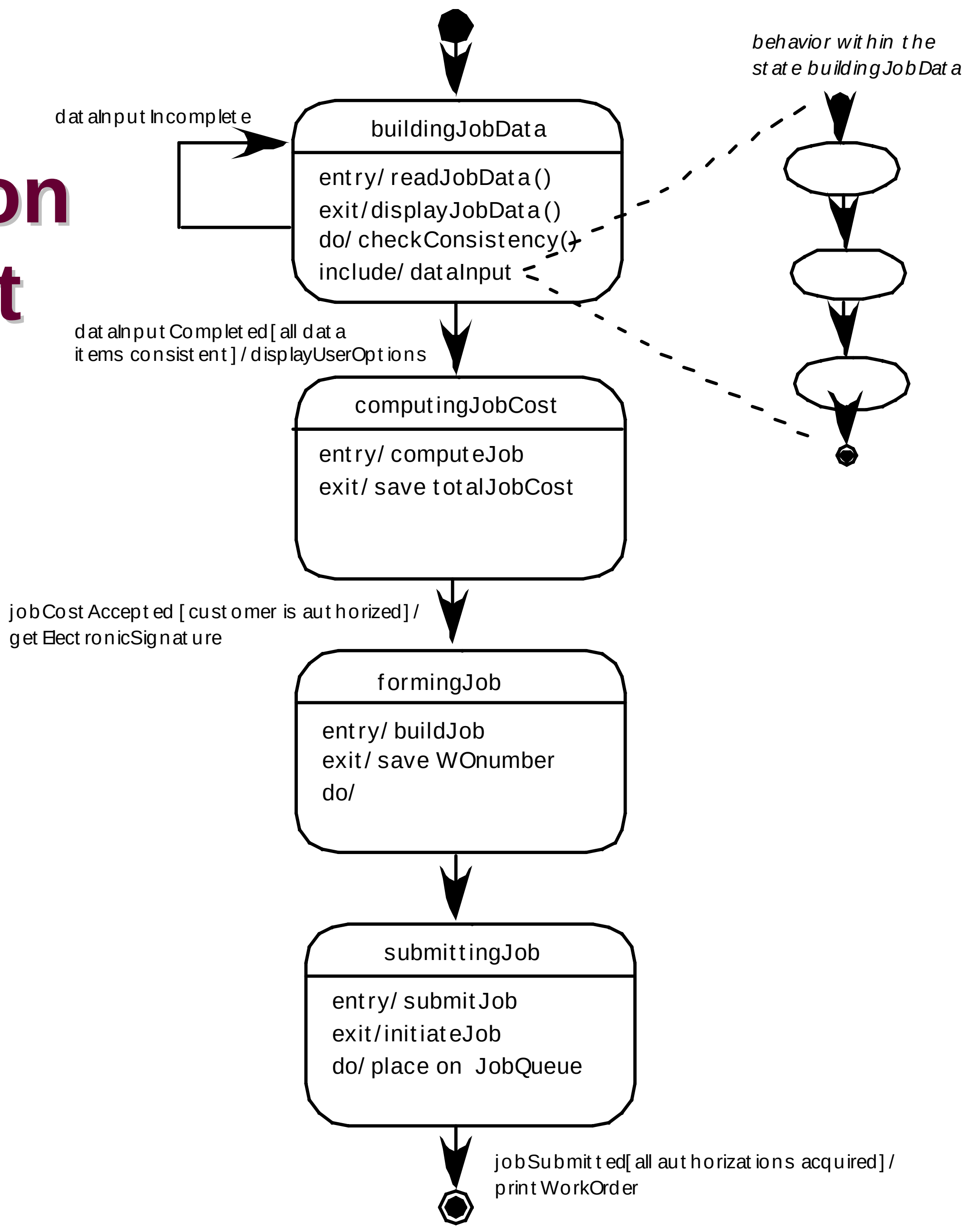
Collaboration Diagram



Processing Flow in 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)

Behavioral Representation in Statechart



Object Constraint Language (OCL)

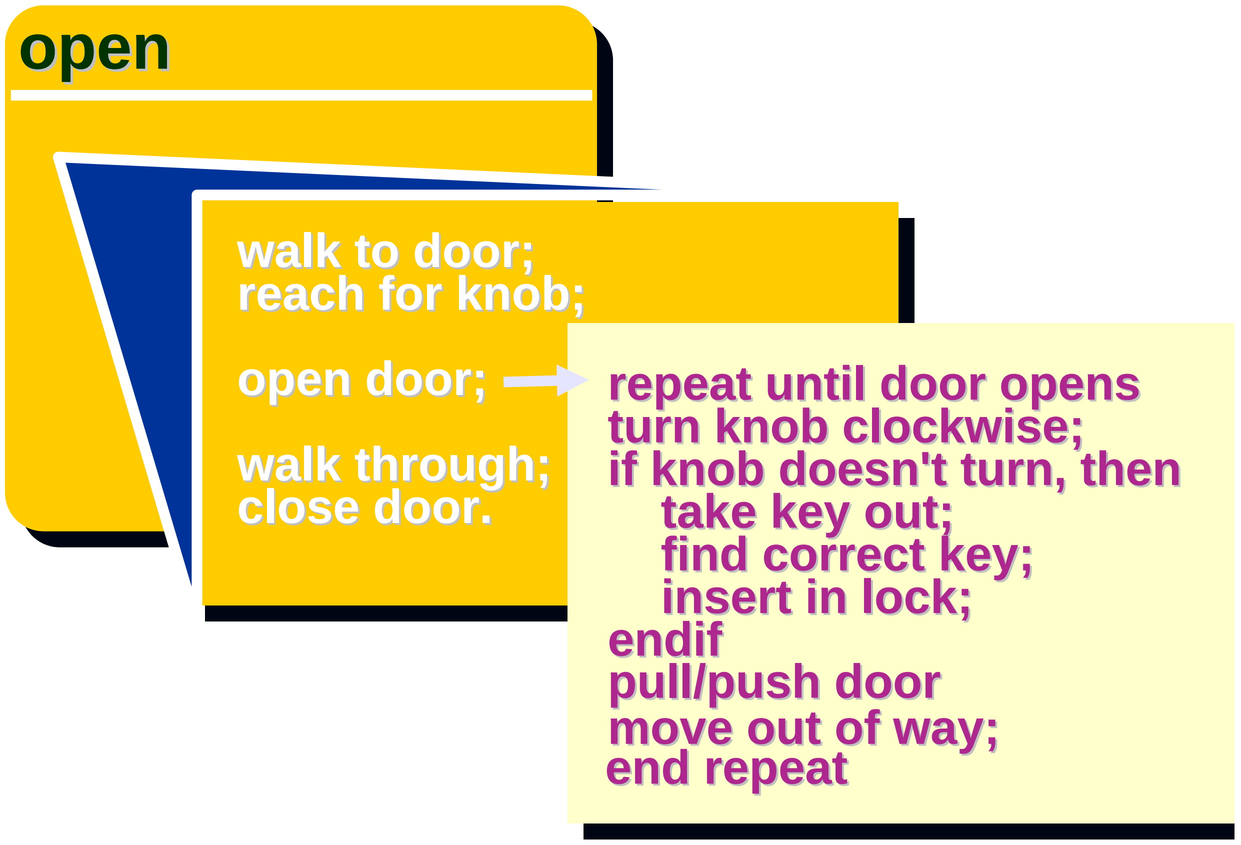
- complements UML by allowing a software engineer to use a formal grammar and syntax to construct unambiguous statements about various design model elements
- simplest OCL language statements are constructed in four parts:
  - (1) a *context* that defines the limited situation in which the statement is valid;
  - (2) a *property* that represents some characteristics of the context (e.g., if the context is a class, a property might be an attribute)
  - (3) an *operation* (e.g., arithmetic, set-oriented) that manipulates or qualifies a property, and
  - (4) keywords (e.g., if, then, else, and, or, not, implies) that are used to specify conditional expressions.

OCL Example

```
context PrintJob::validate(upperCostBound :
Integer, custDeliveryReq :
    Integer)
  pre:  upperCostBound > 0
      and custDeliveryReq > 0
      and self.jobAuthorization = 'no'
  post: if self.totalJobCost <= upperCostBound
        and self.deliveryDate <= custDeliveryReq
      then
        self.jobAuthorization = 'yes'
      endif
```

Algorithm Design

- the closest design activity to coding
- the approach:
  - review the design description for the component
  - use stepwise refinement to develop algorithm
  - use structured programming to implement procedural logic
  - use 'formal methods' to prove logic



```
walk to door;
reach for knob;

open door;

walk through;
close door.
```

```
repeat until door opens
turn knob clockwise;
if knob doesn't turn, then
    take key out;
    find correct key;
    insert in lock;
endif
pull/push door
move out of way;
end repeat
```

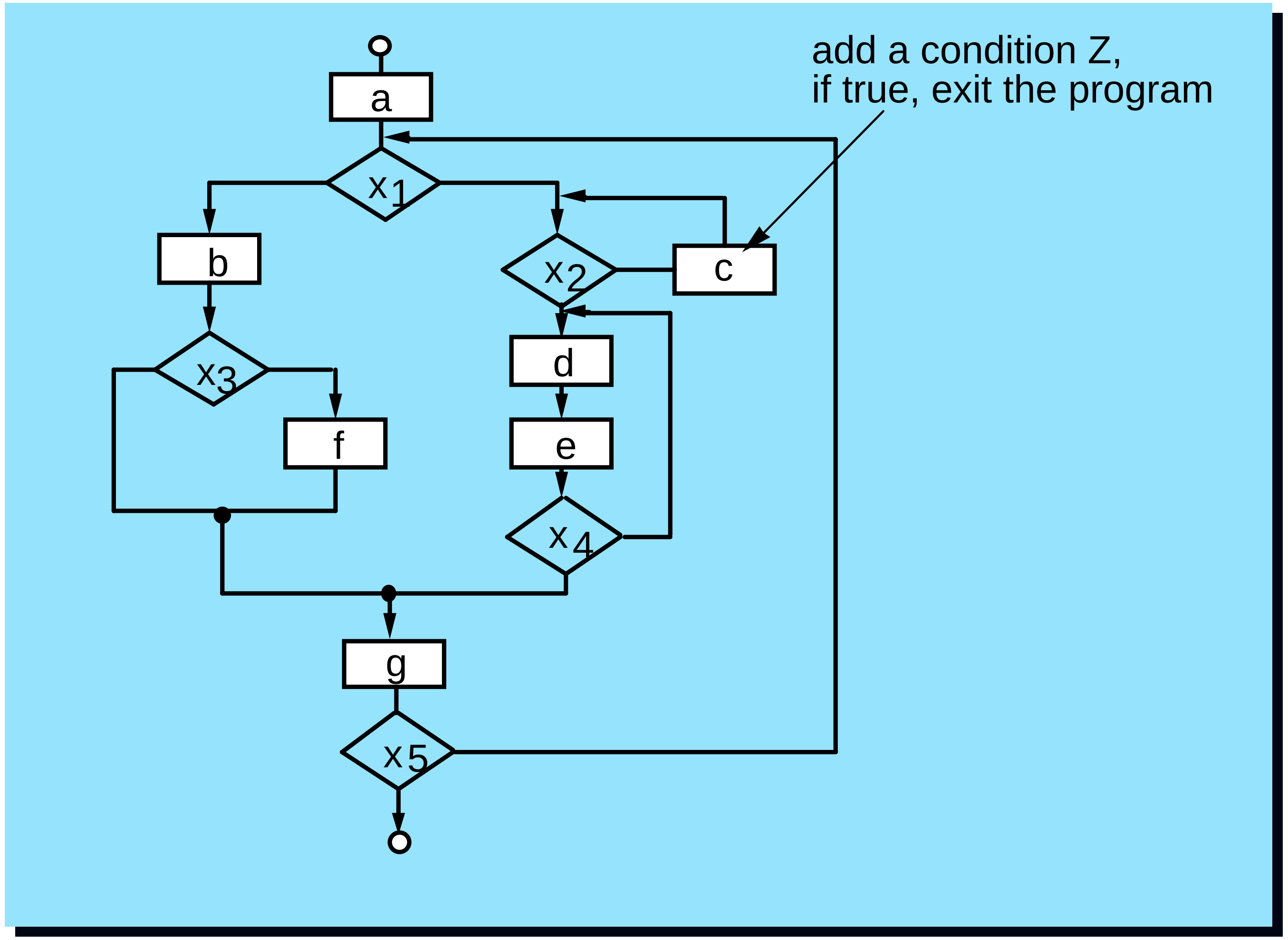
Algorithm Design Model

- represents the algorithm at a level of detail that can be reviewed for quality
- options:
  - graphical (e.g. flowchart, box diagram)
  - pseudocode (e.g., PDL) ... choice of many
  - programming language
  - decision table
  - conduct walkthrough to assess quality

Structured Programming for Procedural Design

- uses a limited set of logical constructs:
  - *sequence*
  - *conditional*— if-then-else, select-case
  - *loops*— do-while, repeat until
- leads to more readable, testable code
- can be used in conjunction with ‘proof of correctness’
- important for achieving high quality, but not enough

A Structured Procedural Design

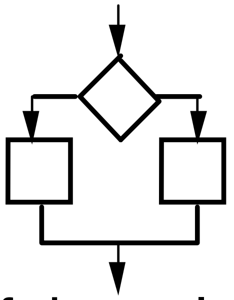


add a condition Z,
if true, exit the program

Decision Table

Conditions	Rules 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
- Can be represented in great detail
- Machine readable, no need for graphics input
- Graphics can be generated from PDL
- Enables declaration of data as well as procedure
- Easier to review and maintain

```
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);
        •
        •
        •
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.timesecon
                        PARBEGIN
                        CALL alarm PROCEDURE WITH "on", alarn
                        CALL phone PROCEDURE WITH message
                        ENDPAR
            ELSE  skip
            ENDIF
        ENDFOR
ENDREP
END security.monitor
```