Lecture 7

- **Design Modelling 2**

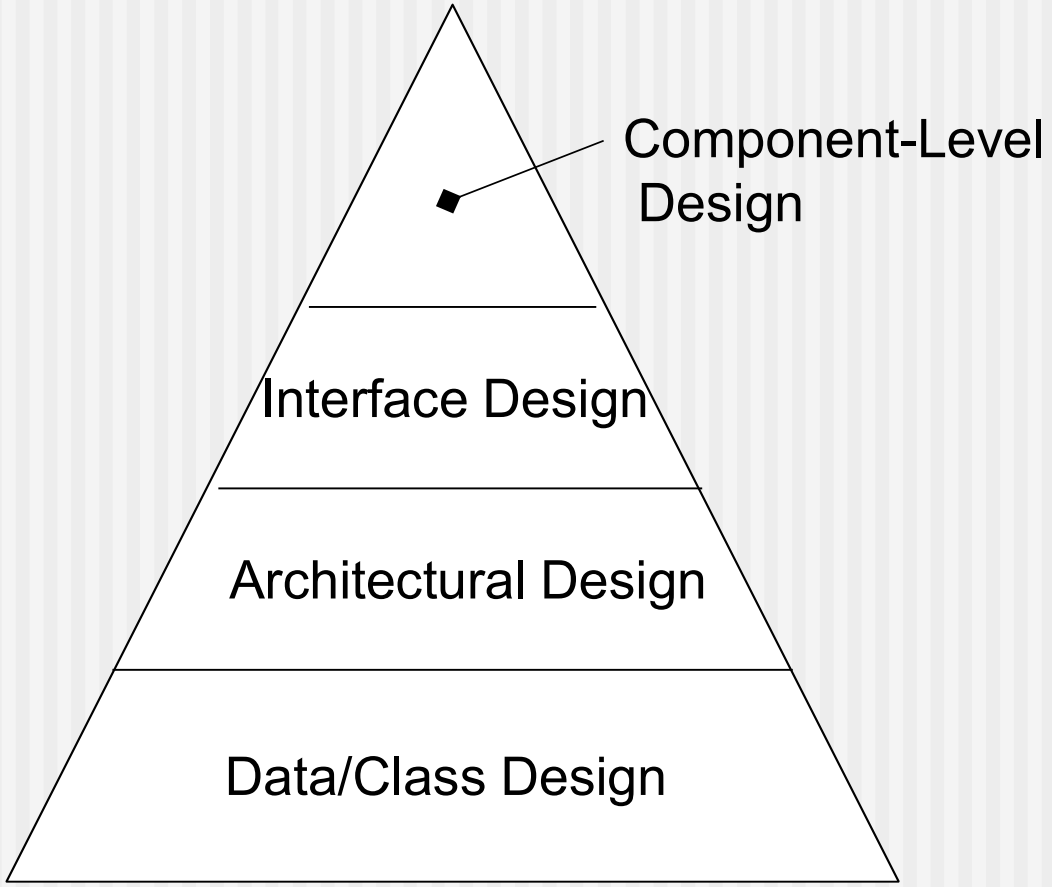
“How to create the representations of the software?”

Topics

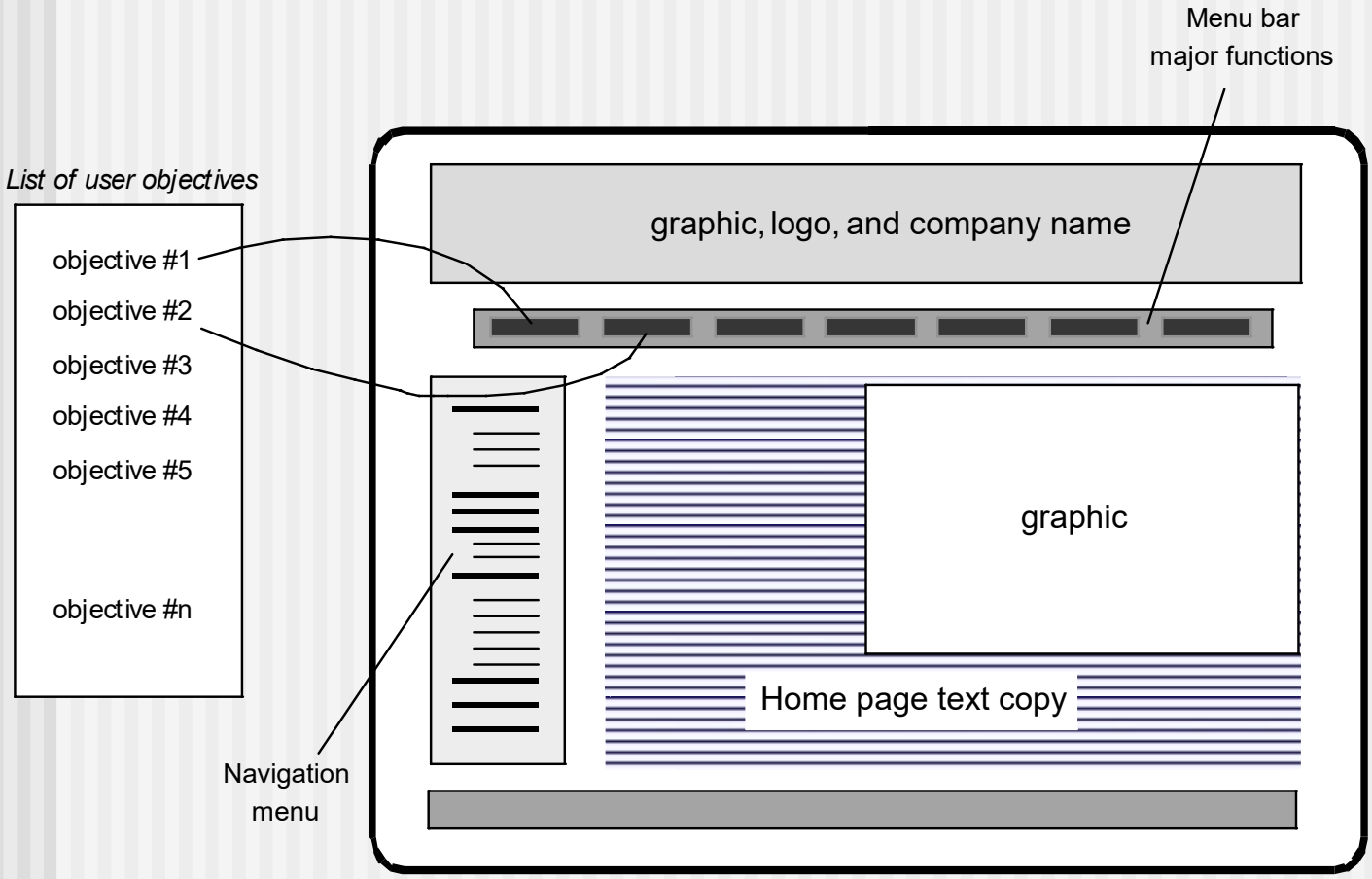
- User Interface Design
- Component-Level Design
- Algorithm Design
- Component-Based Development

1. User Interface Design

The Design Model



Interface Design Example



Interface Design

Easy to learn?

Easy to use?

Easy to understand?

Interface Design

Typical design errors include:

- **Lack of consistency;**
- **Too much memorization;**
- **No guidance / help;**
- **No context sensitivity;**
- **Poor response;**
- **Arcane / unfriendly;**

Golden Rules

- Place the user in control;
- Reduce the user's memory load;
- Make the interface consistent;

Place the User in Control

- Define interaction modes in a way that does not force a user into unnecessary or undesired actions.
- Provide for flexible interaction.
- Allow user interaction to be interruptible and undoable.
- Streamline interaction as skill levels advance and allow the interaction to be customized.
- Hide technical internals from the casual user.
- Design for direct interaction with objects that appear on the screen.

Reduce the User's Memory Load

- Reduce demand on short-term memory.
- Establish meaningful defaults.
- Define shortcuts that are intuitive.
- The visual layout of the interface should be based on a real world metaphor.
- Disclose information in a progressive fashion.

Make the Interface Consistent

- Allow the user to put the current task into a meaningful context.
- Maintain consistency across a family of applications.
- If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.

Interface Analysis

Interface analysis means understanding of:

- The **people** (end-users) who will interact with the system through the interface;
- The **tasks** that end-users must perform to do their work;
- The **content** that is presented as part of the interface, and
- The **environment** in which these tasks will be conducted.

Interface Design Issues

- Response Time
- Help Facilities
- Error Handling
- Menu and Command Labeling
- Application Accessibility
- Internationalization

2. Component-Level Design

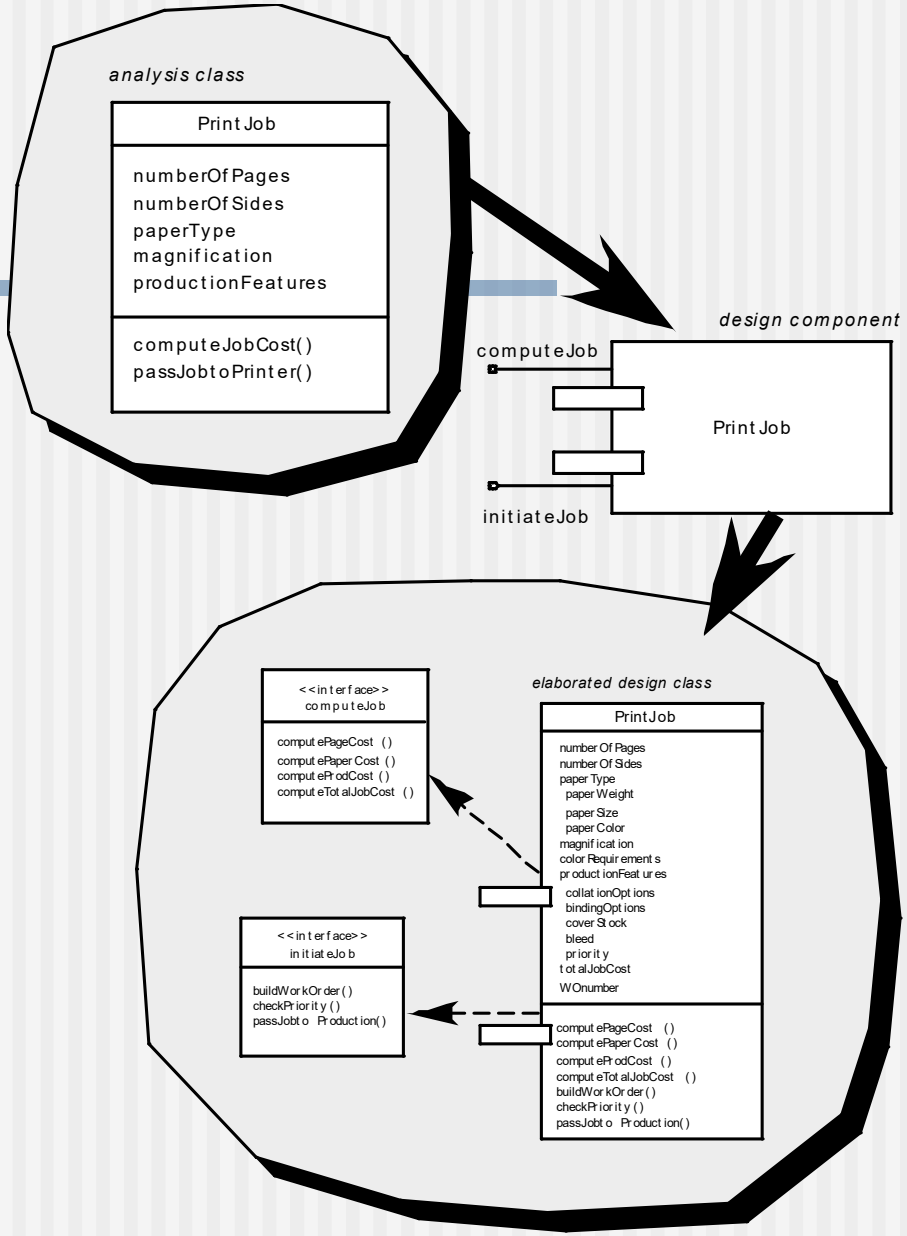
What is a Component?

- OMG Unified Modeling Language Specification defines a component as:

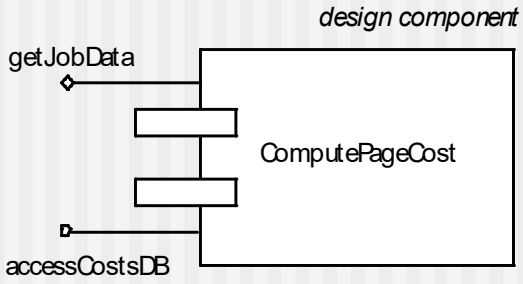
  "… **a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.**"

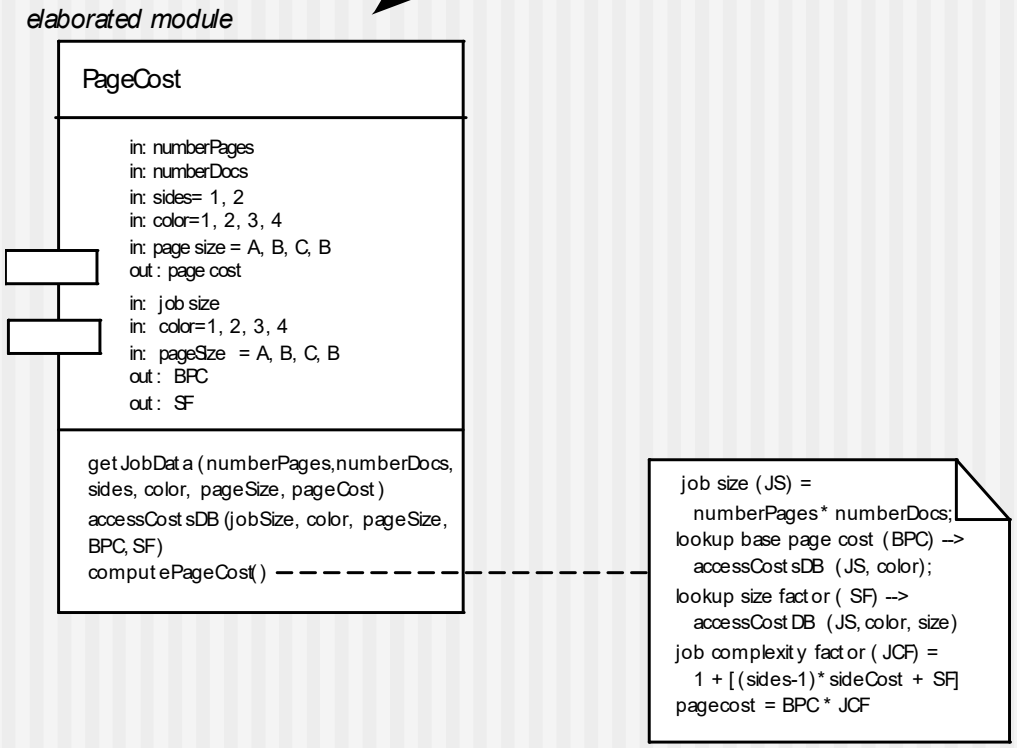
- **OO View** - A component contains a set of collaborating classes.
- **Conventional View** - A component contains processing logic, the internal data structures that are required to implement the processing logic, and an interface that enables the component to be invoked and data to be passed to it.

OO Component



Conventional 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."*

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

- **Conventional View** - The “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

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

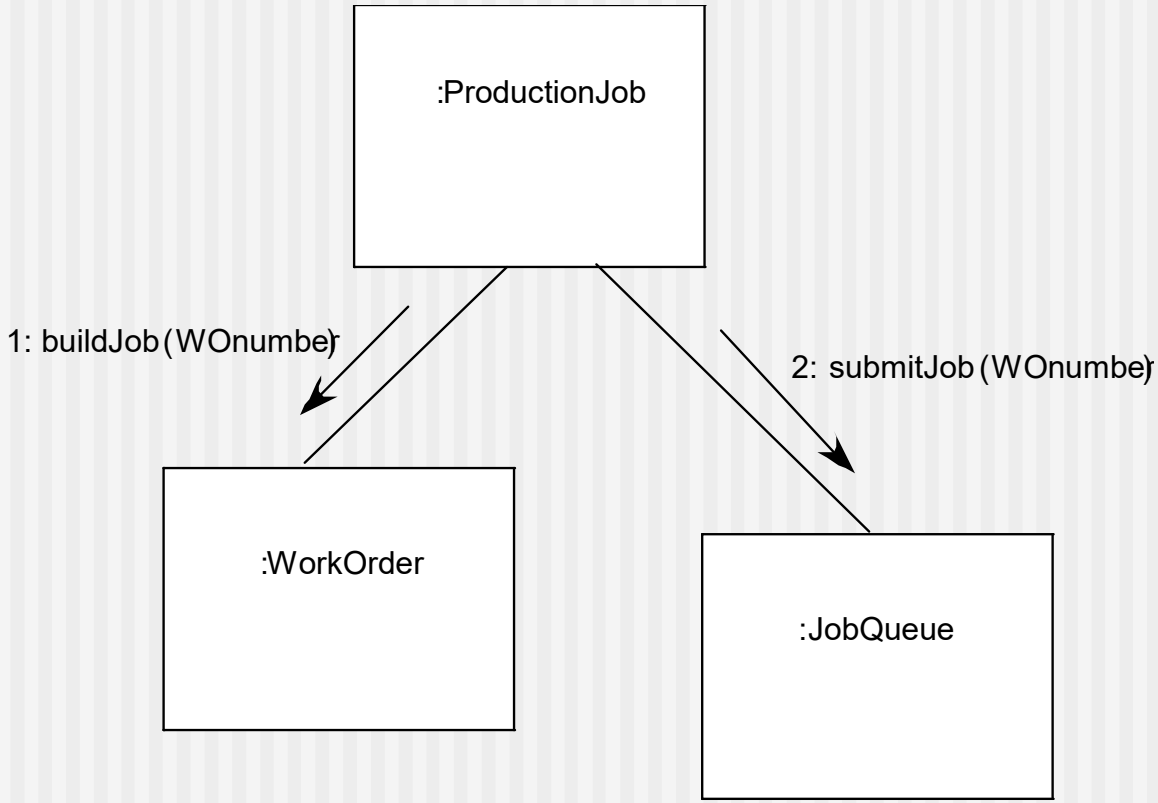
Component-Level Design

- **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 within each operation in detail.

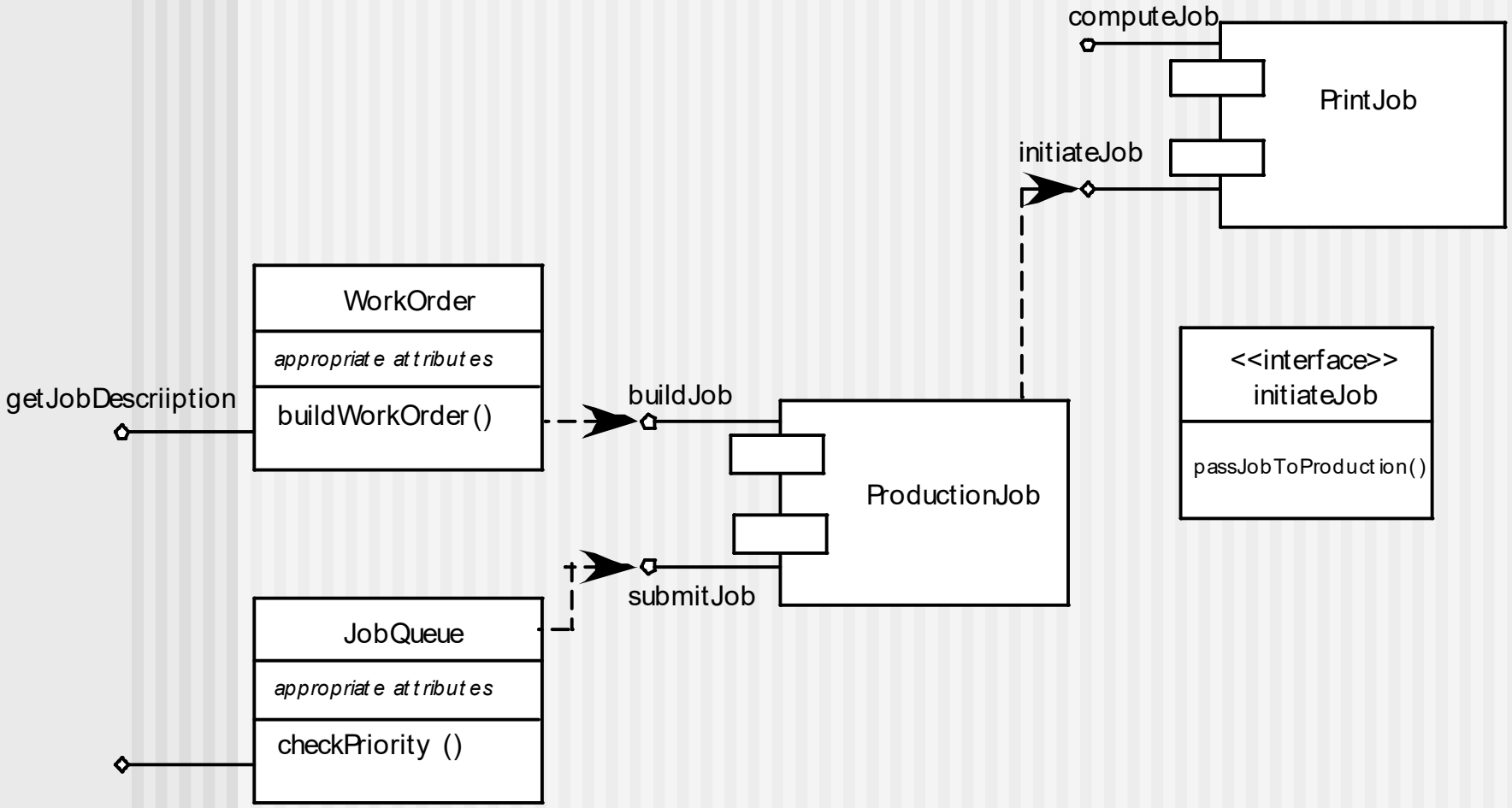
Component-Level Design

- **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



Designing Conventional Components

- The design of processing logic is governed by the basic principles of algorithm design and structured programming
- The design of data structures is defined by the data model developed for the system
- The design of interfaces is governed by the collaborations that a component must effect

3. Algorithm Design

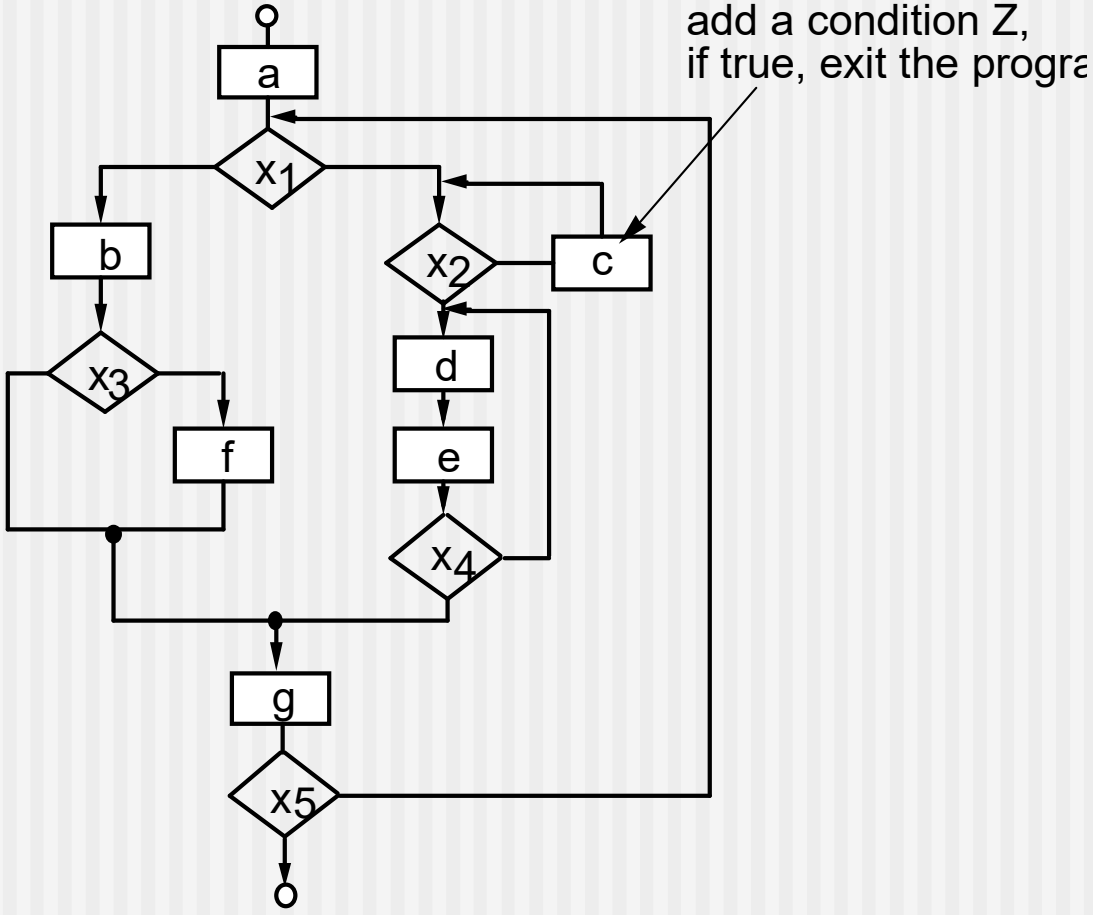
Algorithm Design

- 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.

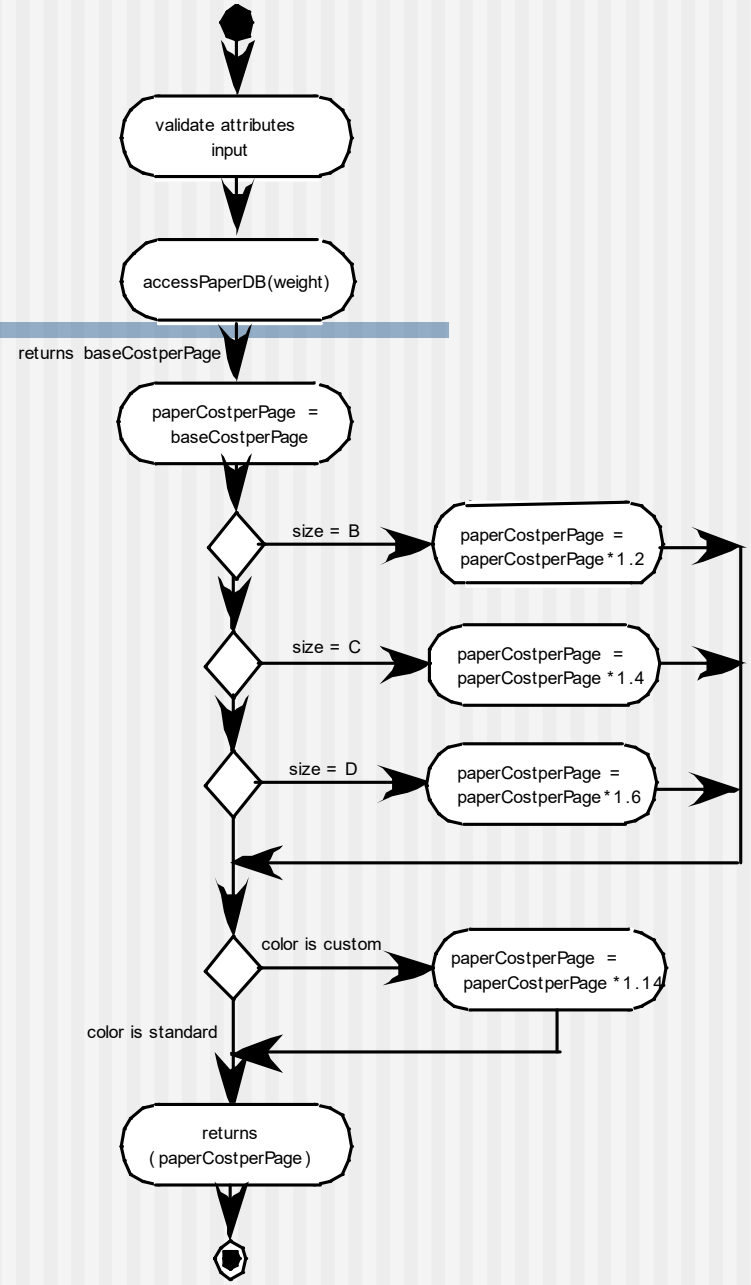
Algorithm Design Model

- Represents the algorithm at a level of detail that can be reviewed for quality.
- Options:
  - **Graphical Design Notation** (e.g. flowchart, box diagram, activity diagram)
  - **Tabular Design Notation** (e.g. decision table)
  - **Program Design Language** (e.g. pseudocode)

A Structured Procedural Design



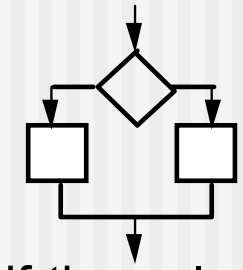
Activity Diagram



Decision Table

	Rules					
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

4. Component-Based Development

Component-Based Development

- When faced with the possibility of reuse, the software team asks:
  - Are commercial off-the-shelf (COTS) components available to implement the requirement?
  - Are internally-developed reusable components available to implement the requirement?
  - Are the interfaces for available components compatible within the architecture of the system to be built?
- At the same time, they are faced with the following impediments to reuse ...

Impediments to Reuse

- Few companies and organizations have anything that even slightly resembles a comprehensive software reusability plan.
- Although an increasing number of software vendors currently sell tools or components that provide direct assistance for software reuse, the majority of software developers do not use them.
- Relatively little training is available to help software engineers and managers understand and apply reuse.
- Many software practitioners continue to believe that reuse is "more trouble than it's worth."
- Many companies continue to encourage of software development methodologies which do not facilitate reuse.
- Few companies provide an incentives to produce reusable program components.

Component Qualification

Before a component can be used, you must consider:

- Application programming interface.
- Development and integration tools required by the component.
- Run-time requirements including resource usage (e.g., memory or storage), timing or speed, and network protocol.
- Service requirements including operating system interfaces and support from other components.
- Security features including access controls and authentication protocol.
- Embedded design assumptions including the use of specific numerical or non-numerical algorithms.
- Exception handling.

Component Adaptation

The implication of “easy integration” is:

- That consistent methods of resource management have been implemented for all components in the library;
- That common activities such as data management exist for all components, and
- That interfaces within the architecture and with the external environment have been implemented in a consistent manner.

Component Composition

- An infrastructure must be established to bind components together.
- Architectural ingredients for composition include:
  - Data Exchange Model
  - Automation
  - Structured Storage
  - Underlying Object Model