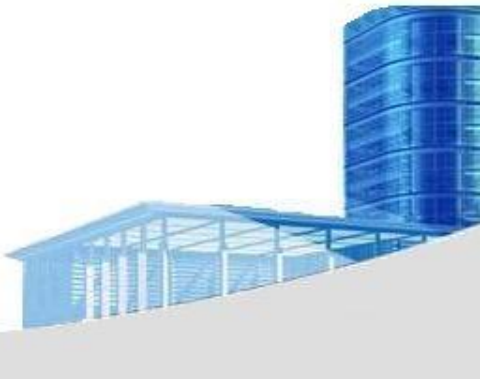
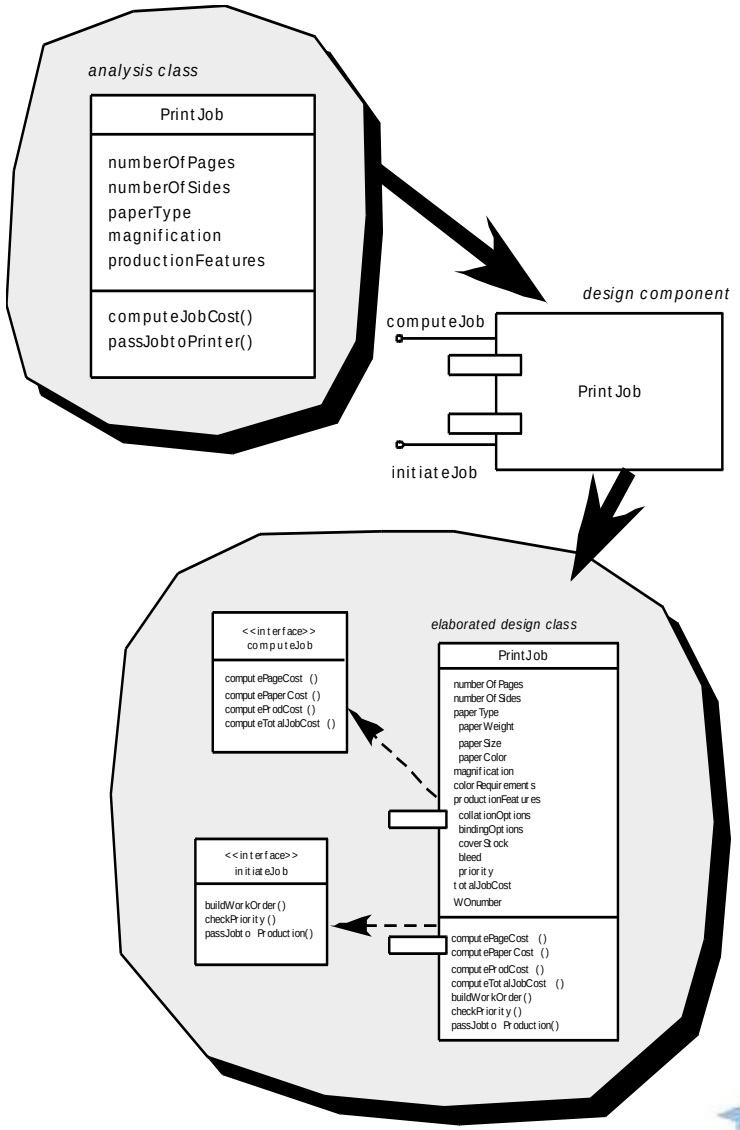
Ch.14 Component-Level Design

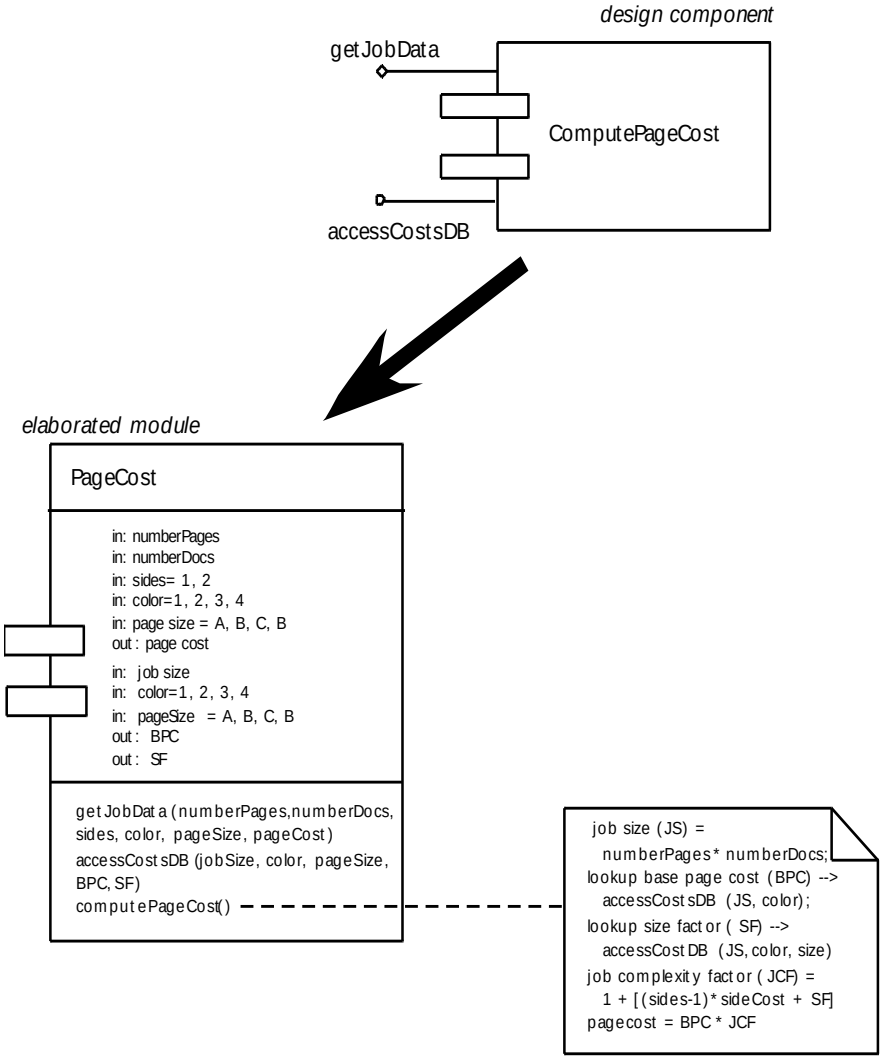


- **What is a Component?**
  - OMG Unified Modeling Language Specification [OMG01] defines a component as
    - "… a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."
  - *OO view*: a component contains a set of collaborating classes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

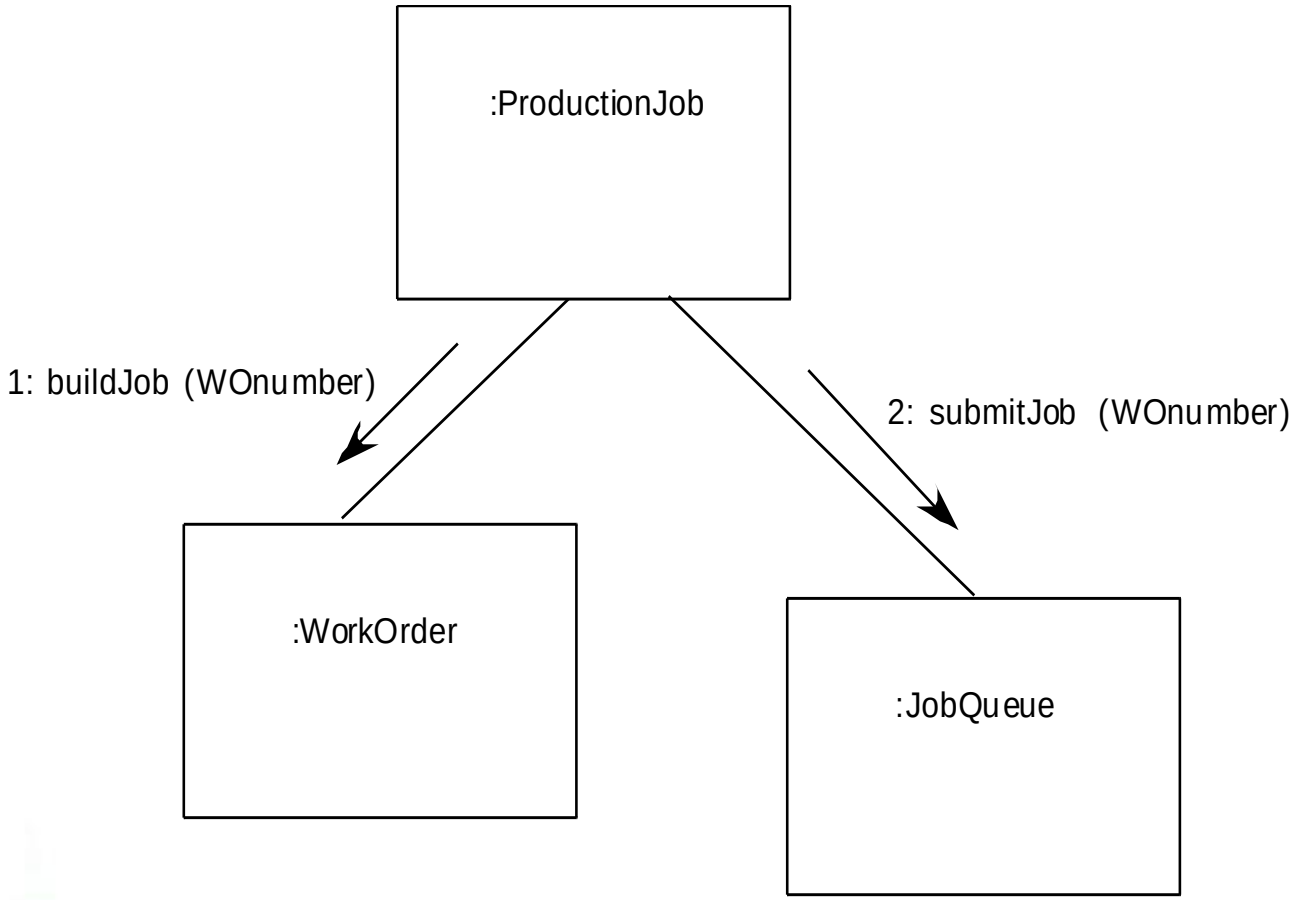
- **Coupling**
  - Conventional view:
    - The degree to which a component is connected to other components and to the external world
  - OO view:
    - a qualitative measure of the degree to which classes are connected to one another
  - Level of coupling
    - Content
    - Common
    - Control
 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

- **Component Level Design - I**

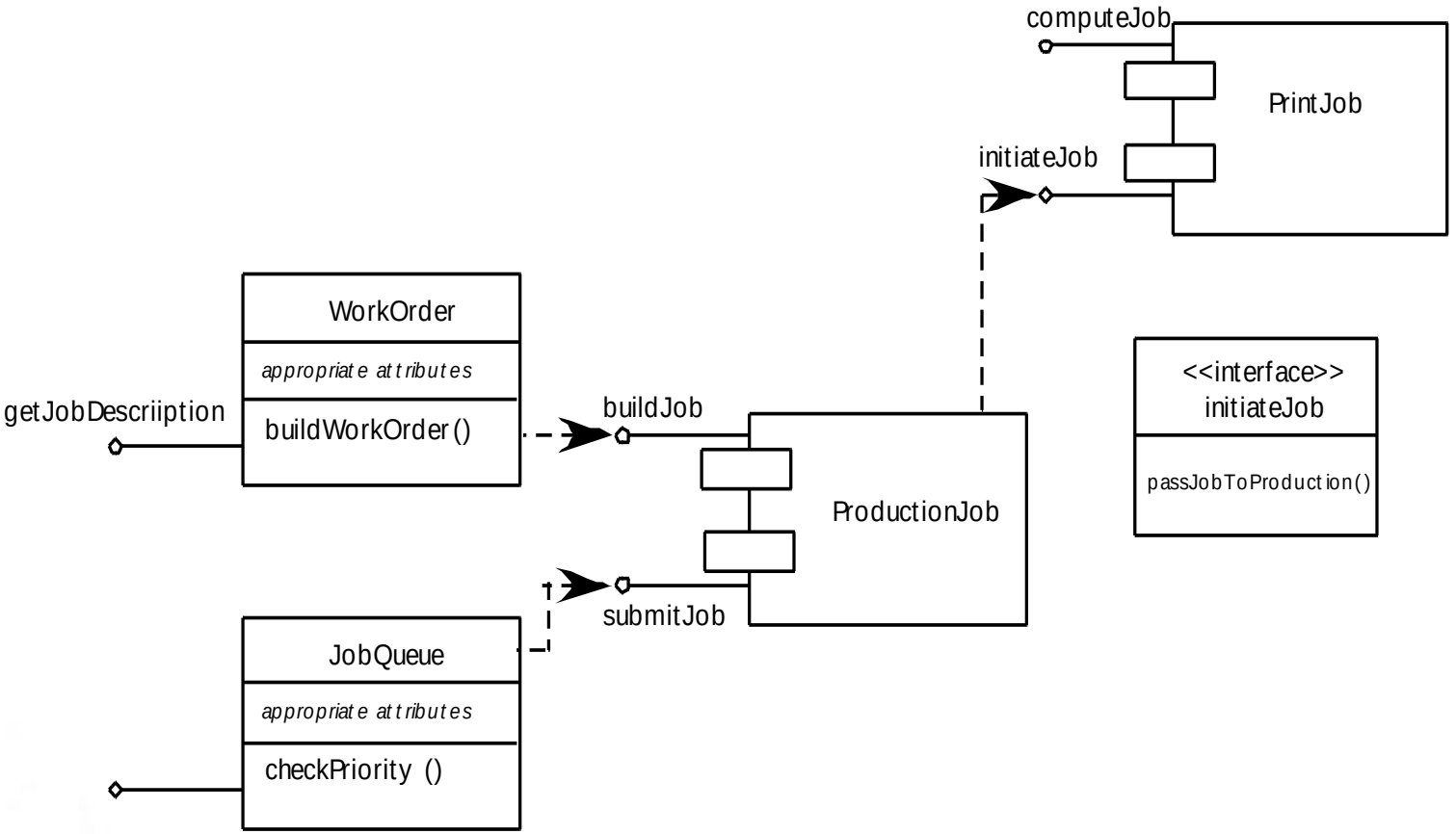
  - Step 1. Identify all design classes that correspond to the problem domain.
  - Step 2. Identify all design classes that correspond to the infrastructure domain.
  - Step 3. Elaborate all design classes that are not acquired as reusable components.
  - Step 3a. Specify message details when classes or component collaborate.
  - Step 3b. Identify appropriate interfaces for each component.

- **Component Level Design - II**
  - 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.
  - 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.

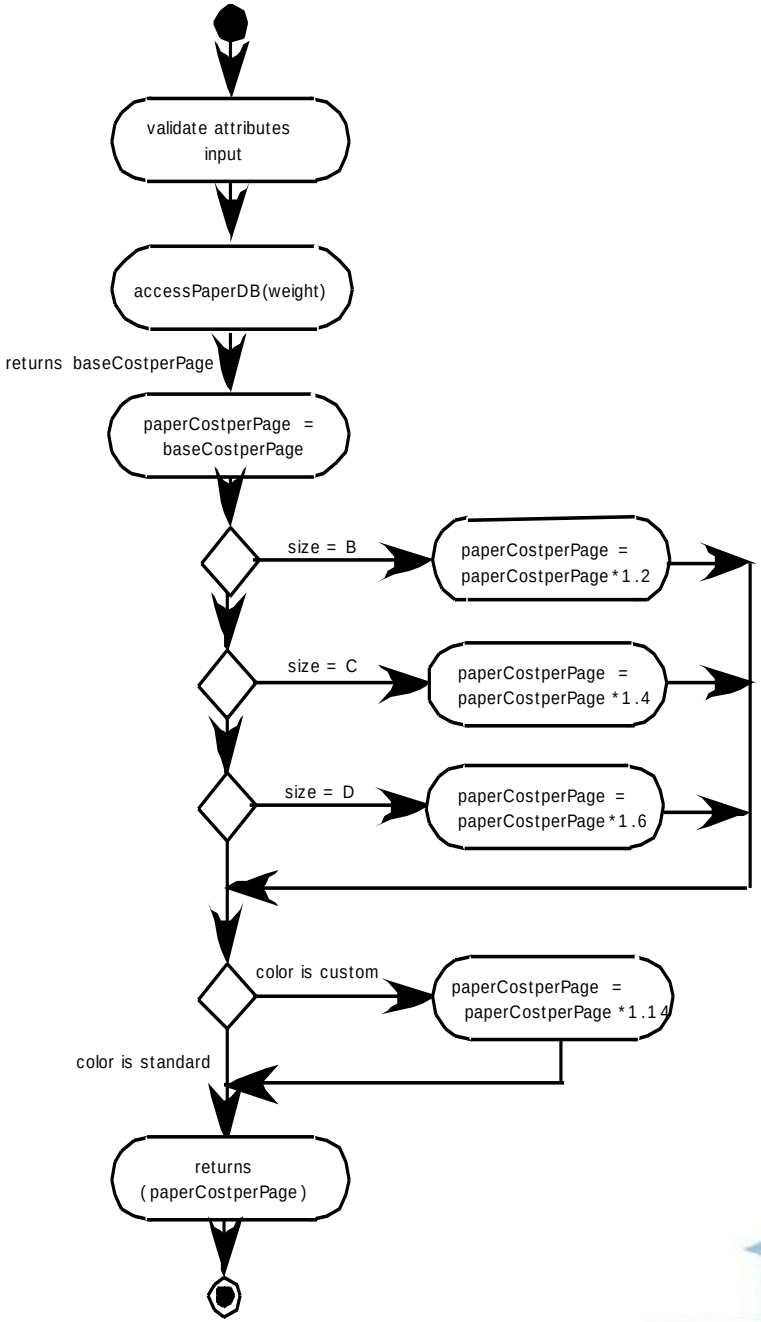
- **Collaboration Diagram**



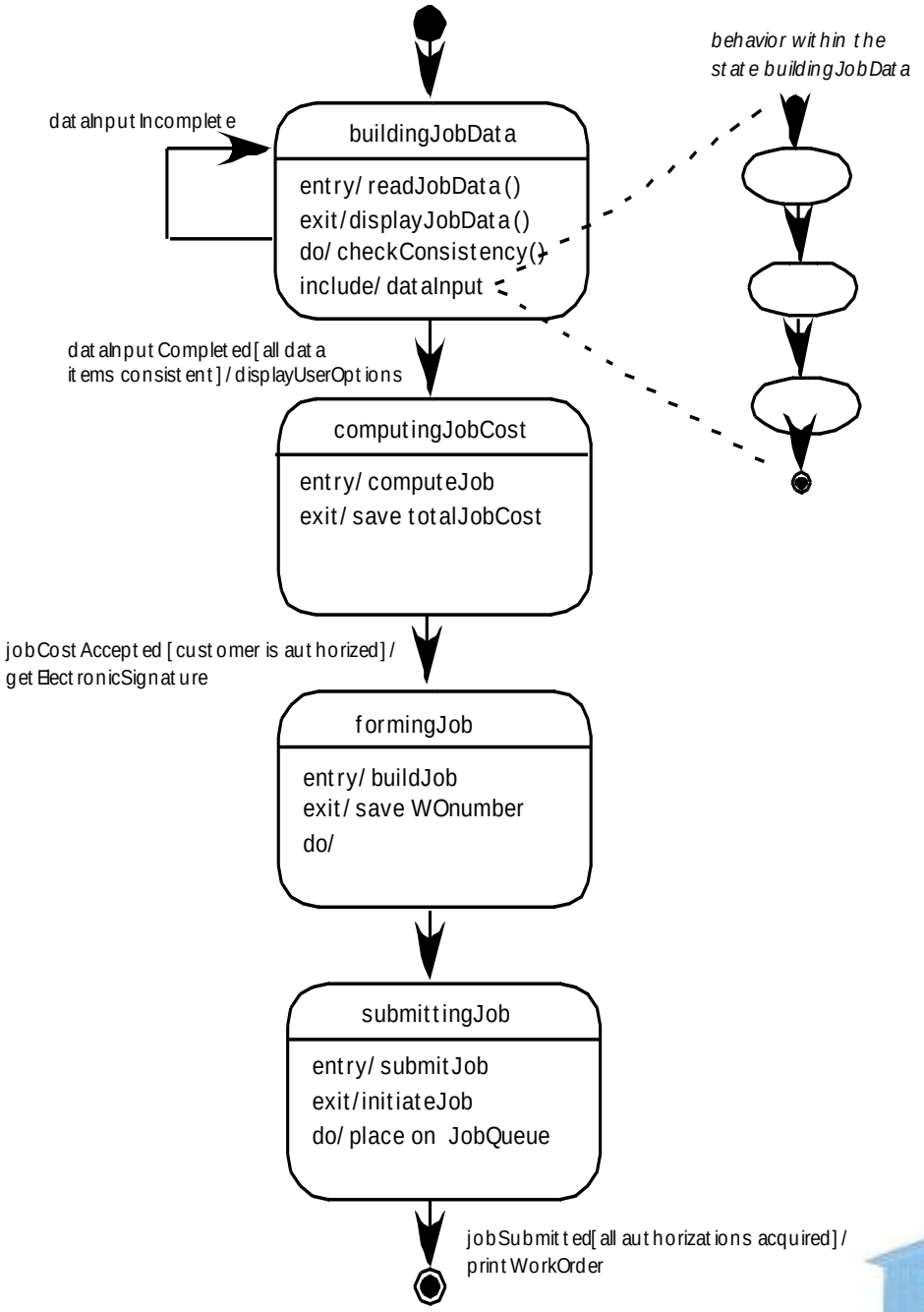
- **Refactoring**



- **Activity Diagram**



- **Statechart**


behavior within the state buildingJobData
dataInput Incomplete
buildingJobData
entry/ readJobData()
exit/displayJobData()
do/ checkConsistency()
include/ dataInput
dataInput Completed[all data items consistent]/displayUserOptions
computingJobCost
entry/ computeJob
exit/ save totalJobCost
jobCost Accepted [customer is authorized]/ getElectronicSignature
formingJob
entry/ buildJob
exit/ save WOnumber
do/
submittingJob
entry/ submitJob
exit/initiateJob
do/ place on JobQueue
jobSubmitted[all authorizations acquired]/ printWorkOrder

- **Component Design for WebApps**

  - WebApp component is
    - (1) a well-defined cohesive function that manipulates content or provides computational or data processing for an end-user, or
    - (2) a cohesive package of content and functionality that provides end-user with some required capability.
  - Therefore, component-level design for WebApps often incorporates elements of content design and functional design.

- **Content Design for WebApps**
  - focuses on content objects and the manner in which they may be packaged for presentation to a WebApp end-user
  - consider a Web-based video surveillance capability within SafeHomeAssured.com
    - potential content components can be defined for the video surveillance capability:
      - (1) the content objects that represent the space layout (the floor plan) with additional icons representing the location of sensors and video cameras;
      - (2) the collection of thumbnail video captures (each an separate data object), and
      - (3) the streaming video window for a specific camera.
    - Each of these components can be separately named and manipulated as a package.

- **Functional Design for WebApps**

  - Modern Web applications deliver increasingly sophisticated processing functions that:
    - (1) perform localized processing to generate content and navigation capability in a dynamic fashion;
    - (2) provide computation or data processing capability that is appropriate for the WebApp's business domain;
    - (3) provide sophisticated database query and access, or
    - (4) establish data interfaces with external corporate systems.
  - To achieve these (and many other) capabilities, you will design and construct WebApp functional components that are identical in form to software components for conventional software.

- **Functional Design for WebApps**

  - Thin web-based client
    - Interface layer only on device
    - Business and data layers implemented using web or cloud services
  - Rich client
    - All three layers (interface, business, data) implemented on device
    - Subject to mobile device limitations

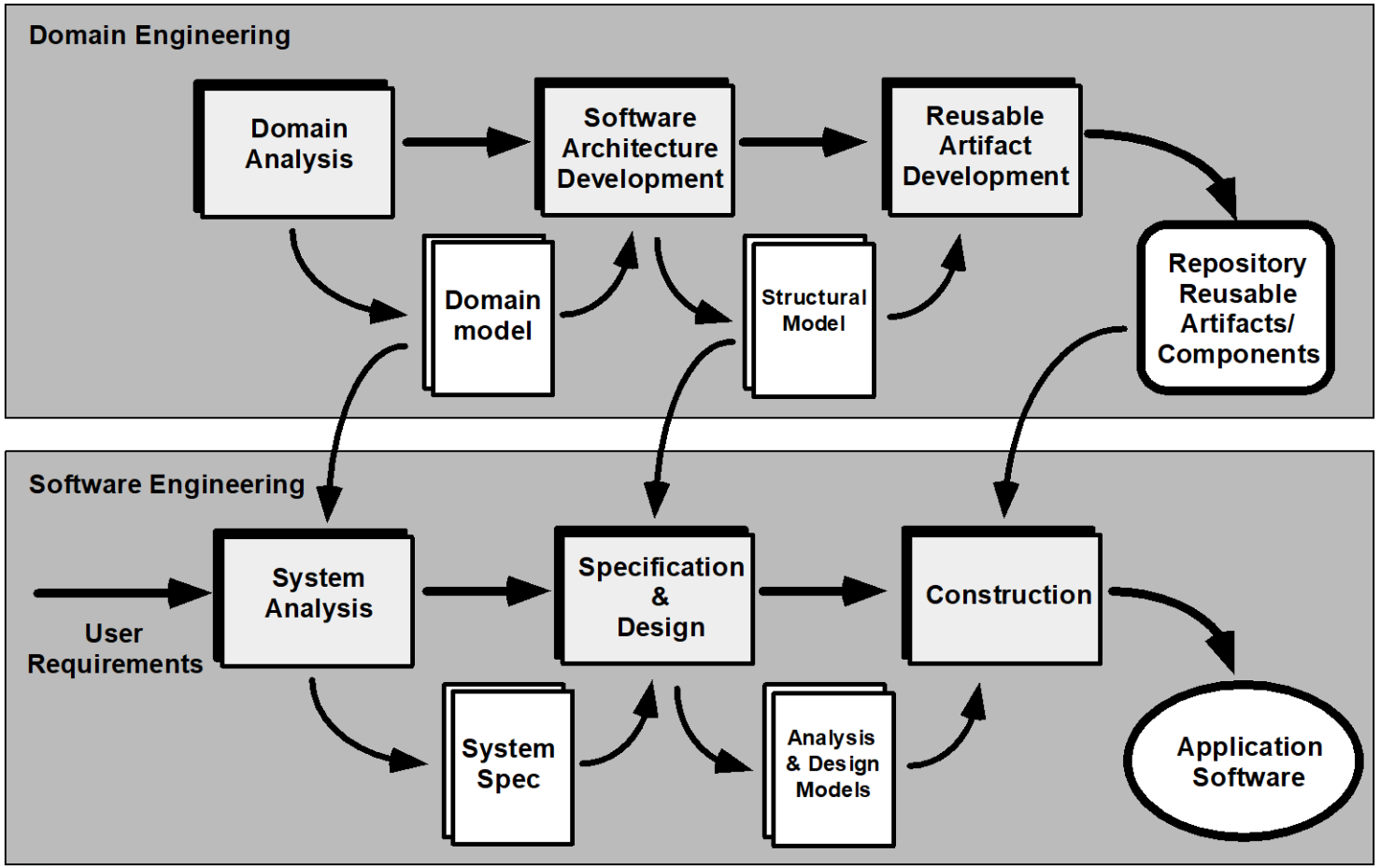
- **Designing Conventional Components**

  - 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

- **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.

- **The CBSE Process**



- **Domain Engineering**

  - 1. Define the domain to be investigated.
  - 2. Categorize the items extracted from the domain.
  - 3. Collect a representative sample of applications in the domain.
  - 4. Analyze each application in the sample.
  - 5. Develop an analysis model for the objects.

- **Identifying Reusable Components**
  - Is component functionality required on future implementations?
  - How common is the component's function within the domain?
  - Is there duplication of the component's function within the domain?
  - Is the component hardware-dependent?
  - Does the hardware remain unchanged between implementations?
  - Can the hardware specifics be removed to another component?
  - Is the design optimized enough for the next implementation?
  - Can we parameterize a non-reusable component so that it becomes reusable?
  - Is the component reusable in many implementations with only minor changes?
  - Is reuse through modification feasible?
  - Can a non-reusable component be decomposed to yield reusable components?
  - How valid is component decomposition for reuse?

- **Component-Based SE**
  - a library of components must be available
  - components should have a consistent structure
  - a standard should exist, e.g.,
    - OMG/CORBA
    - Microsoft COM
    - Sun JavaBeans

- **CBSE Activities**

  - Component qualification
  - Component adaptation
  - Component composition
  - Component update

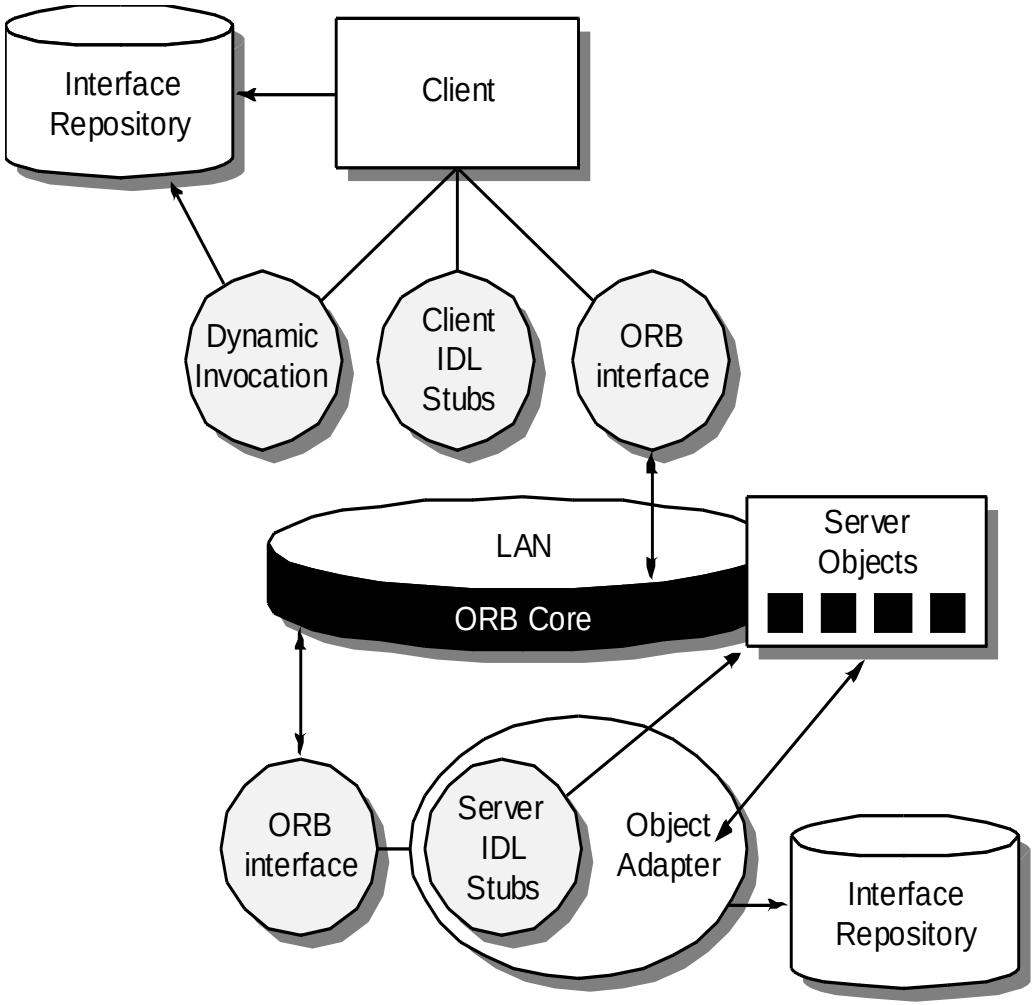
- **Qualification**
  - *Mitch Kapor, the creator of Lotus 1-2-3, presented a "software design manifesto" in Dr. Dobbs Journal. He said:*
    - application programming interface (API)
    - development and integration tools required by the component
    - run-time requirements including resource usage (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

- **Adaptation**
  - *The implication of “easy integration” is:*
    - (1) that consistent methods of resource management have been implemented for all components in the library;
    - (2) that common activities such as data management exist for all components, and
    - (3) that interfaces within the architecture and with the external environment have been implemented in a consistent manner.

- **Composition**
  - An infrastructure must be established to bind components together
  - Architectural ingredients for composition include:
    - Data exchange model
    - Automation
    - Structured storage
    - Underlying object model

- **OMG/ CORBA**
  - The Object Management Group has published a *common object request broker architecture* (OMG/CORBA).
  - An object request broker (ORB) provides services that enable reusable components (objects) to communicate with other components, regardless of their location within a system.
  - Integration of CORBA components (without modification) within a system is assured if an interface definition language (IDL) interface is created for every component.
  - Objects within the client application request one or more services from the ORB server. Requests are made via an IDL or dynamically at run time.
  - An interface repository contains all necessary information about the service's request and response formats.

- **ORB Architecture**



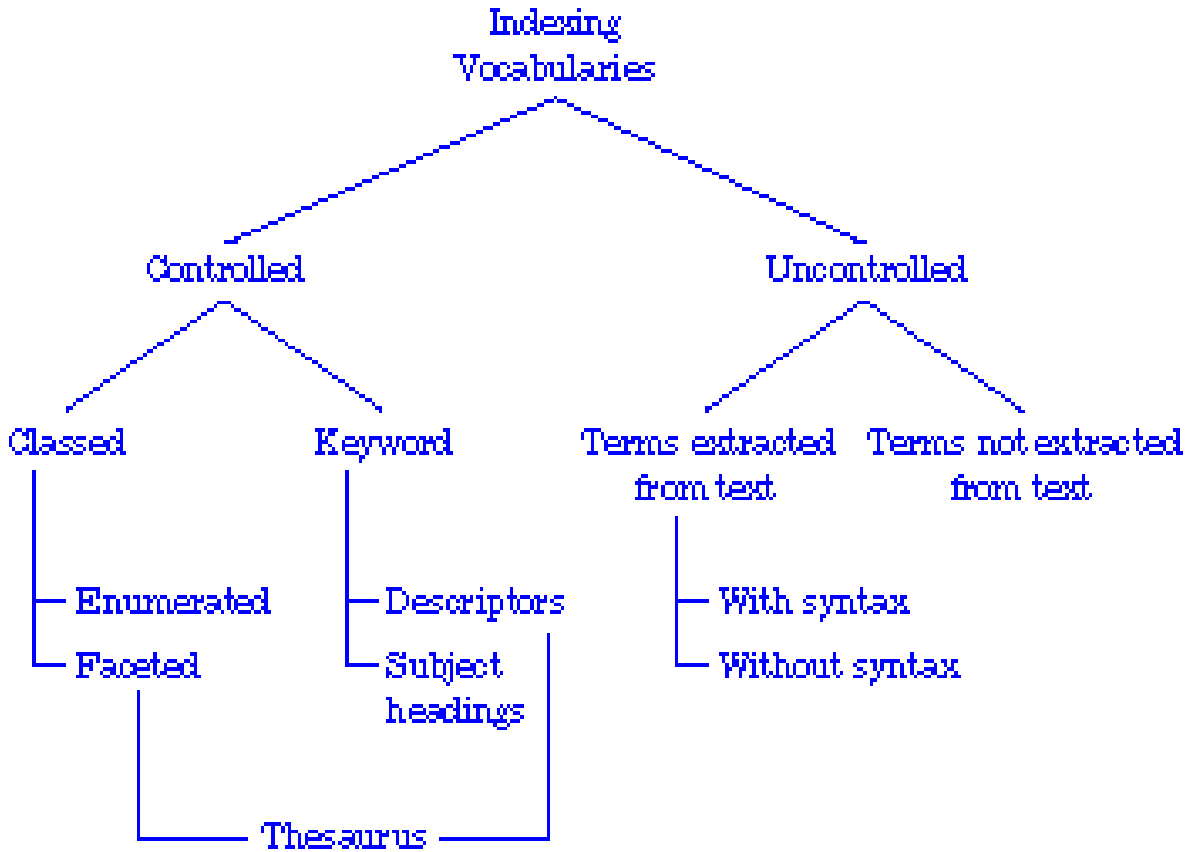
- **Microsoft COM**
  - The component object model (COM) provides a specification for using components produced by various vendors within a single application running under the Windows operating system.
  - COM encompasses two elements:
    - COM interfaces (implemented as COM objects)
    - a set of mechanisms for registering and passing messages between COM interfaces.

- **Sun JavaBeans**
  - The JavaBeans component system is a portable, platform independent CBSE infrastructure developed using the Java programming language.
  - The JavaBeans component system encompasses a set of tools, called the Bean Development Kit (BDK), that allows developers to
    - analyze how existing Beans (components) work
    - customize their behavior and appearance
    - establish mechanisms for coordination and communication
    - develop custom Beans for use in a specific application
    - test and evaluate Bean behavior.

- **Classification**

  - *Enumerated classification*—components are described by defining a hierarchical structure in which classes and varying levels of subclasses of software components are defined
  - *Faceted classification*—a domain area is analyzed and a set of basic descriptive features are identified
  - *Attribute-value classification*—a set of attributes are defined for all components in a domain area

- **Indexing**


Indexing Vocabularies
Controlled
Uncontrolled
Classed
Keyword
Terms extracted from text
Terms not extracted from text
Enumerated
Faceted
Descriptors
Subject headings
With syntax
Without syntax
Thesaurus

- **The Reuse Environment**
  - A component database capable of storing software components and the classification information necessary to retrieve them.
  - A library management system that provides access to the database.
  - A software component retrieval system (e.g., an object request broker) that enables a client application to retrieve components and services from the library server.
  - CBSE tools that support the integration of reused components into a new design or implementation.