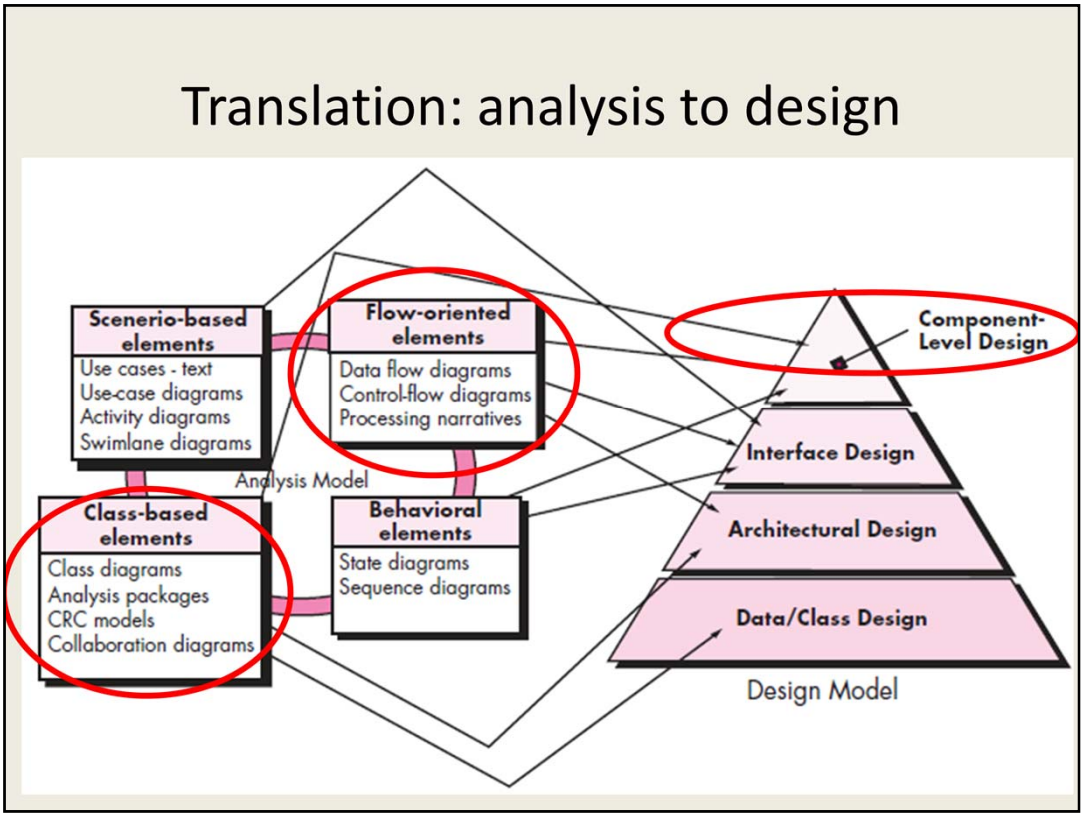
# Component level design
# (III)




Data/Class design: transforms analysis classes into implementation classes and data structures
Architectural design: defines relationships among the major structural elements of the software, the architectural styles and patterns
Interface design–defines how software elements, hardware elements, and end-users communicate. Usage scenarios and behavioral models are used
*Component-level design–transforms structural elements into procedural descriptions of software components. Class-based models and behavioral models server as the basis*

Only scenario-based elements serve just one design model

## Designing traditional components

- Fundamental constructs
  - Sequential
  - Condition
  - Repetition
- Structured programming
  - A design technique for component level design
    - Constrain logic flows to these three constructs
  - Limit structural design to small number of predictable logical structure
    - Reduce program complexity
    - Enhance readability, testability, and maintainability
    - Conducive to human understanding --- Chunking

The foundations of component-level design for traditional software components were formed in the early 1960s and were solidified with the work of Edsger Dijkstra and his colleagues; In the late 1960s, Dijkstra and others proposed the use of a set of constrained logical constructs from which any program could be formed. The constructs emphasized "maintenance of functional domain." ; That is, each construct had a predictable logical structure and was entered at the top and exited at the bottom, enabling a reader to follow procedural flow more easily.

*Sequence* implements processing steps that are essential in the specification of any algorithm.

*Condition* provides the facility for selected processing based on some logical occurrence, and

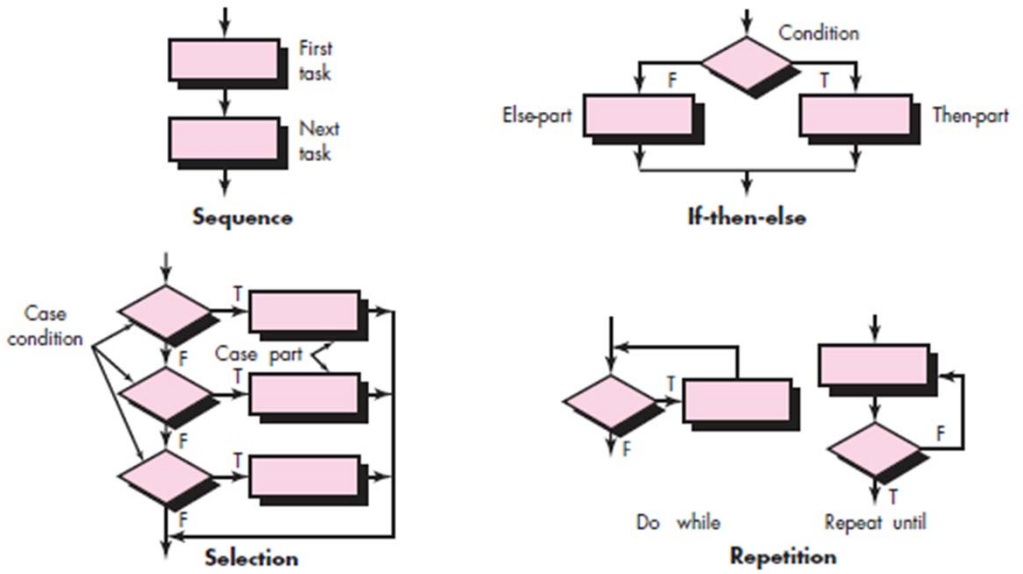
*Repetition* allows for looping.

These three constructs are fundamental to *structured programming*—an important component-level design technique.

The structured constructs are **logical chunks** that allow a reader to recognize procedural elements of a module, rather than reading the design or code line by

line. Understanding is enhanced when readily recognizable logical patterns are encountered.

# Designing traditional components: graphical notations (flowchart)




The activity diagram allows you to represent sequence, condition, and repetition—

all elements of structured programming—and is a descendent of an earlier pictorial

design representation (still used widely) called a *flowchart.* A flowchart, like

an activity diagram, is quite simple pictorially. A box is used to indicate a processing

step. A diamond represents a logical condition, and arrows show the flow

of control.

The *selection* (or *select-case*) construct shown in the figure

is actually an extension of the *if-then-else.*

# Designing traditional components: tabular notations (decision table)

- Express complex combination of conditions
- Steps
  - **1.** List all actions that can be associated with a specific procedure (or component).
  - **2.** List all conditions (or decisions made) during execution of the procedure.
  - **3.** Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions; alternatively, develop every possible permutation of conditions.
  - **4.** Define rules by indicating what actions occur for a set of conditions.

| | 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 |
| **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.

# Designing traditional components: program design language (PDL)

- Logical constructs + natural language
- NOT a programming language
- Example
  - SafeHome security

```
component alarmManagement;
The intent of this component is to manage control panel switches and input from sensors by
type and to act on any alarm condition that is encountered.
    set default values for systemStatus (returned value), all data items
    initialize all system ports and reset all hardware
    check controlPanelSwitches (cps)
        if cps = "test" then invoke alarm set to "on"
        if cps = "alarmOff" then invoke alarm set to "off"
        if cps = "newBoundingValue" then invoke keyboardInput
        if cps = "burglarAlarmOff" invoke deactivateAlarm;
        .
        .
        .
        default for cps = none
    reset all signalValues and switches
    do for all sensors
        invoke checkSensor procedure returning signalValue
        if signalValue > bound [alarmType]
            then phoneMessage = message [alarmType]
                set alarmBell to "on" for alarmTimeSeconds
                set system status = "alarmCondition"
                parbegin
                    invoke alarm procedure with "on", alarmTimeSeconds;
                    invoke phone procedure set to alarmType, phoneNumber
                endpar
            else skip
        endif
    enddofor
end alarmManagement
```

*Program design language* (PDL), also called *structured English* or *pseudocode,* incorporates

the logical structure of a programming language with the free-form expressive

ability of a natural language.

It should be noted that PDL can be extended to include keywords for multitasking and/or concurrent processing, interrupt

handling, interprocess synchronization, and many other features.

## Component-based development

- CBSE (Component-based Software Engineering)
  - a process that emphasizes the design and construction of computer-based systems using reusable software "components."
- COTS (Commercial off-the-shelf) components
  - purchase of packaged solutions which are then adapted to satisfy the needs of the purchasing organisation

In the software engineering context, reuse is an idea both old and new.

Programmers have reused ideas, abstractions, and processes since the earliest days of computing, but the early approach to reuse was ad hoc

Today, complex, high-quality computer based systems must be built in very short time periods and demand a more organized approach to reuse.

# Component-Based Development

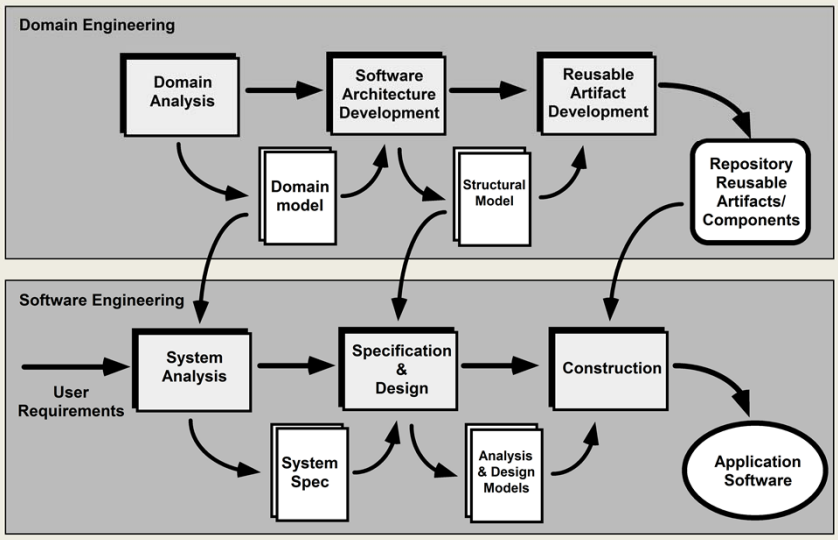
- When faced with the possibility of reuse, the software team asks:
  - Are commercial off-the-shelf (COTS) components available to implement the requirement?
  - Are internally-developed reusable components available to implement the requirement?
  - Are the interfaces for available components compatible 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-based development

- Software process for CBSE



## Component-based development

- 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.

*Domain engineering provides the library of reusable components that are required for component-based software engineering.*

The intent of *domain engineering* is to identify, construct, catalog, and disseminate a set of software components that have applicability to existing and future software in a particular application domain

Overall goal: establish mechanisms that enable software engineers to share these components—to reuse them—during work on new and existing systems.

Domain engineering includes three major activities—analysis, construction, and dissemination.

# 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 development

- the existence of reusable components does not guarantee that these components can be integrated easily or effectively into the architecture chosen for a new application.
  - Qualification
  - Adaptation
  - Composition

Some of these reusable components are developed in-house, others can be extracted from existing applications, and still

others may be acquired from third parties.

Unfortunately, the existence of reusable components does not guarantee that

these components can be integrated easily or effectively into the architecture chosen

for a new application.

It is for this reason that a sequence of component-based development actions is applied when a component is proposed for use.

## Component-based development

- Component qualification
  - ensure a candidate component is appropriate to adopt
  - Factors to consider
    - API
    - Development and integration tools
    - Run-time requirements
    - timing or speed, and network protocol
    - Service requirements
    - Security features
    - Embedded design assumptions
    - Exception handling.

Component qualification ensures that a candidate component will perform the function required, will properly “fit” into the architectural style (Chapter 9) specified for the system, and will exhibit the quality characteristics(e.g., performance, reliability, usability) that are required for the application.

An interface description provides useful information about the operation and use of a software component, but it does not provide all of the information required to determine if a proposed component can, in fact, be reused effectively in a new application.

• 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 nonnumerical algorithms.

• Exception handling.

it is much more difficult to determine the internal workings of commercial off-the-shelf (COTS) or third-party components

because *the only available information may be the interface specification itself*.

## Component-based development

- Component adaptation
  - Address possible conflicts after qualification
  - Adaptation technique (*component wrapping*)
    - White-box wrapping
    - Gray-box wrapping
    - Black-box wrapping

In reality, even after a component has been qualified for use within an application architecture, conflicts may occur in one or more of the areas just noted.

*White-box wrapping* is applied when a software team has full access to the internal design and code for a component (often not the case unless open-source COTS components are used), white-box wrapping examines the internal processing details of the component and makes code-level modifications to remove any conflict.

*Gray-box wrapping* is applied when the component library provides a component extension language or API that enables conflicts to be removed or masked.

*Black-box wrapping* requires the introduction of pre-and postprocessing at the component interface to remove or mask conflicts.

## Component-based development

- Component composition
  - assembles qualified, adapted, and engineered components to populate the architecture established for an application
  - Need an infrastructure for binding the components
    - OMG/CORBA
    - Microsoft COM, .Net
    - Sun JavaBeans Components

To accomplish this, an infrastructure must be established to bind the components into an operational system.

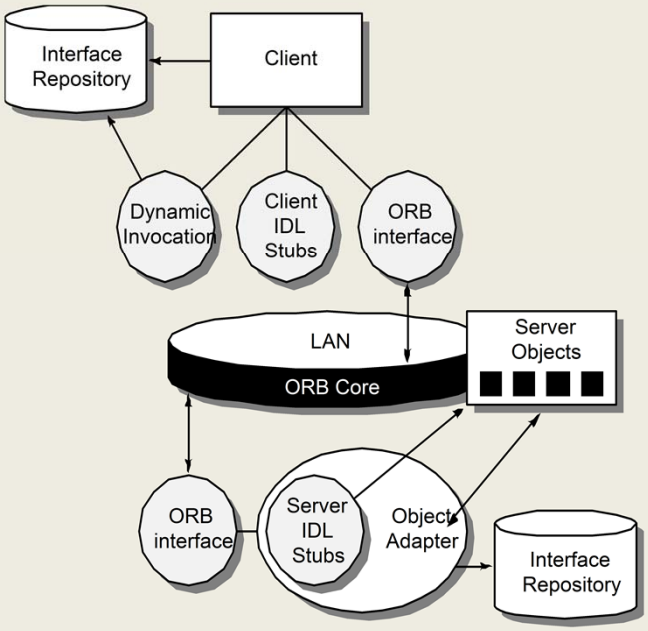
The infrastructure (usually a library of specialized components) provides a model for the coordination of components and specific services that enable components to coordinate with one another and perform common tasks.

# 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.

The Object Management Group has published a *common object request broker architecture* (OMG/CORBA).

# 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.

Microsoft has developed a *component object model* (COM) that provides a specification for using components produced by various vendors within a single application running under the Windows operating system.

## 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.

The JavaBeans component system is a portable, platform-independent CBSE infrastructure developed using the Java programming language.

# Architectural Mismatch

- One of the challenges facing widespread reuse is architectural mismatch
- The designer of reusable components often make implicit assumptions about the environment to which the component is coupled
- These assumptions often focus on the component control model, the nature of the component connections (interfaces), the architectural infrastructure itself, and the nature of the construction process
- If these assumptions are incorrect, architectural mismatch occurs

# Architectural Mismatch

- **All the design concepts contribute to the creation of software components that are reusable and prevent architectural mismatch**
  - Abstraction, hiding, functional independence, refinement, and structured programming, along with object-oriented methods, testing, software quality assurance (SQA), and correctness verification methods),

## Architectural Mismatch

- Early detection of architectural mismatch can occur if stakeholder assumptions are explicitly documented
- The use of a risk-driven process model emphasizes the definition of early architectural prototypes and points to areas of mismatch
- Repairing architectural mismatch is often very difficult without making use of mechanisms like wrappers or adapters
- Sometimes, it is necessary to completely redesign a component interface or the component itself to remove coupling issues

## Analysis/design for reuse

- Specification matching
  - Elements of the requirements model are compared to descriptions of reusable components
  - When matching an existing component, extract from a reuse library and use it
- DFR (Design for Reuse)
  - **Standard data.** If the application domain has standard global data structures, the component should be designed to make use of these standard data structures
  - **Standard interface protocols** within an application domain should be adopted,
  - An architectural style that is appropriate for the domain can serve as a **template** for the architectural design of new software

Although the CBSE process encourages the use of existing software components,

*there are times when new software components must be developed and integrated*

with existing COTS and in-house components. Because these new components

become members of the in-house library of reusable components, they should be

engineered for reuse.

If components cannot be found (i.e., there is no match), a new component is created.

when you begin to create a new component—that *design for reuse* (DFR) should be considered.

a number of key issues that form a basis for design for reuse: standard data, …

## Classifying/retrieving components

- Too many reusable components, how do we search?
- Classification – 3C model
  - Concept, content, and context
  - What, how, and where

Consider a large university library. Hundreds of thousands of books, periodicals, and other information resources are available for use.

But to access these resources, a categorization scheme must be developed.

3C model: A description of **what** the component accomplishes, **how** this is achieved with content that may be hidden from causal users and need be known only to those who intend to modify or test the component, and **where** the component resides within its domain of applicability

## Classifying/retrieving components

- Too many reusable components, how do we search?
- Classification – 3C model
  - Concept, content, and context
  - What, how, and where
- Classification enables you to find and retrieve candidate reusable components, but a reuse environment must exist to integrate these components effectively.

Consider a large university library. Hundreds of thousands of books, periodicals, and other information resources are available for use.

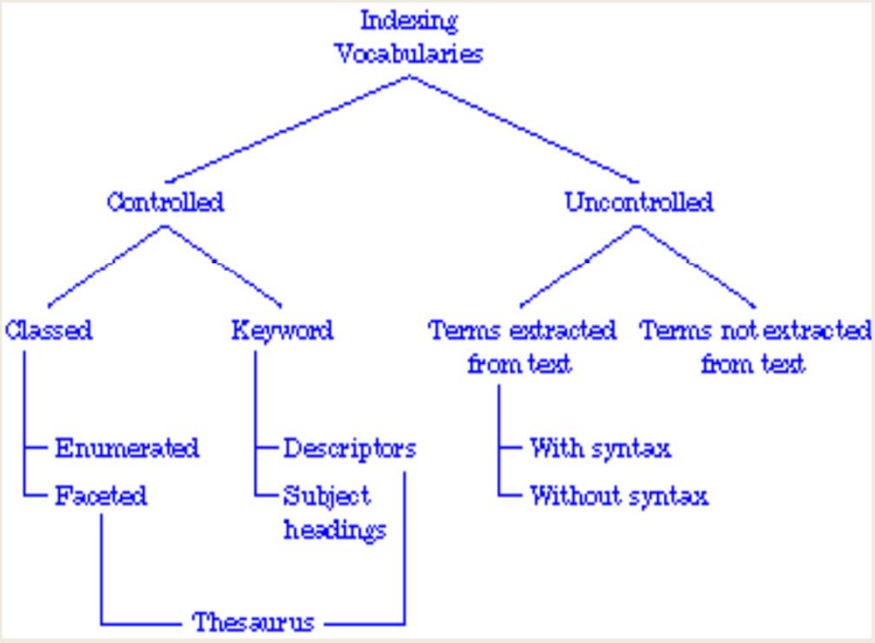
But to access these resources, a categorization scheme must be developed.

3C model: A description of **what** the component accomplishes, **how** this is achieved with content that may be hidden from causal users and need be known only to those who intend to modify or test the component, and **where** the component resides within its domain of applicability

# 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.

The characteristics of a reuse environment are: a component database …

Each of these functions interact with or is embodied within the confines of a **reuse library**, one element of a larger software repository

# Summary

- Designing traditional components
  - Graphical notation
  - Tabular notation
  - PDL
- Component-based development
  - Domain engineering
  - Qualification, adaptation, composition
  - Classification, retrieval