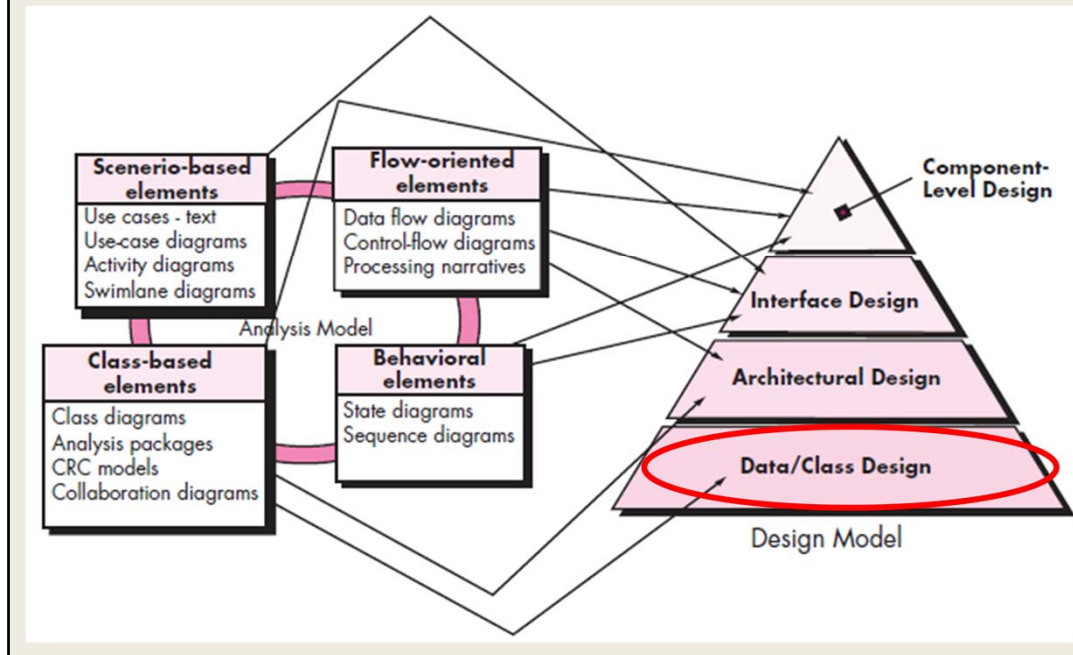


Design Engineering:
process, concepts, and model
(II)



Translation: analysis to design



Now you see why we need different elements in requirements modeling?
Each of the elements of the requirements model (Chapters 6 and 7) provides information that is necessary to create the four design models required for a complete specification of design.

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 serve as the basis

Only scenario-based elements serve just one design model

Data/class design

- Review: OO design concepts
 - Design classes
 - Entity classes (defining things in the physical world) – analysis classes
 - Boundary classes (defining interfaces)
 - Controller classes (defining interactions and transactions).
 - Inheritance - responsibilities of a superclass are immediately inherited by all subclasses
 - Messages
 - stimulate some behavior to occur in the receiving object
 - Implemented as function calls.
 - Polymorphism—a characteristic that greatly reduces the effort required to extend the design
 - E.g., operator overloading

Requirements modeling (also called analysis modeling) focuses primarily on classes that are extracted directly from the statement of the problem----entity classes.

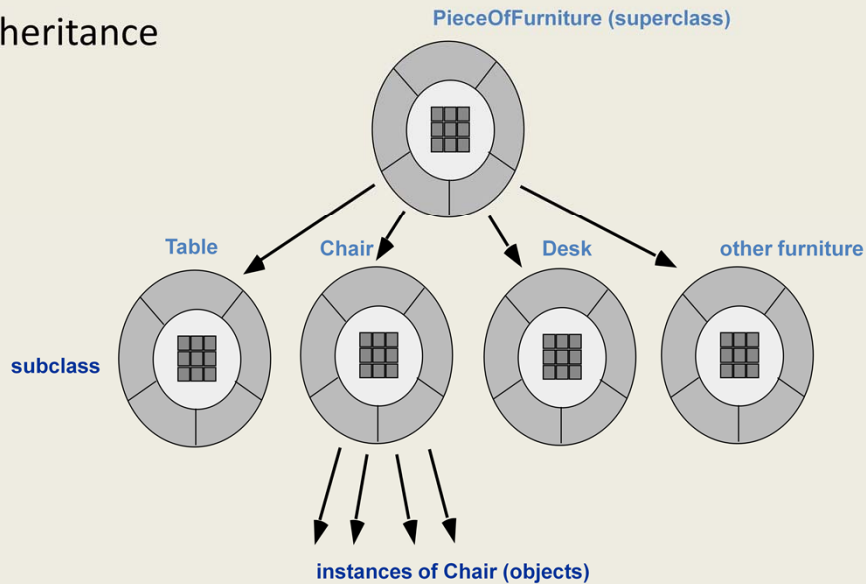
Design refines and extends the set of entity classes. Boundary and controller classes are developed and/or refined during design.

Boundary classes create the interface (e.g., interactive screen and printed reports) that the user sees and interacts with as the software is used.

Controller classes are designed with the responsibility of managing the way entity objects are represented to users.

Data/class design

- Inheritance

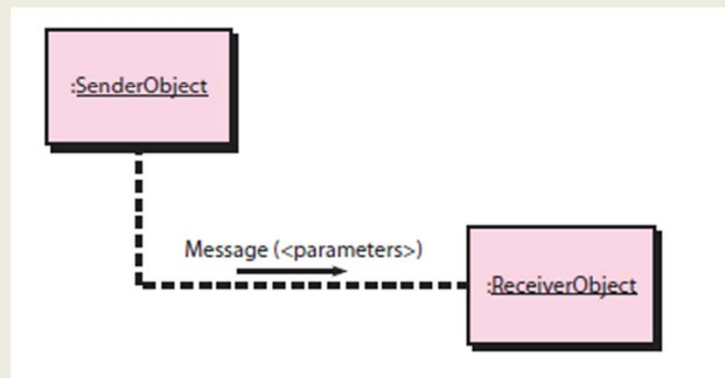


In fact, whenever a new class is to be created, you have a number of options:

- The class can be designed and built from scratch. That is, inheritance is not used.
- The class hierarchy can be searched to determine if a class higher in the hierarchy contains most of the required attributes and operations. The new class inherits from the higher class and additions may then be added, as required.
- The class hierarchy can be restructured so that the required attributes and operations can be inherited by the new class.
- Characteristics of an existing class can be overridden, and different versions of attributes or operations are implemented for the new class.

Data/class design

- Message



A message stimulates some behavior to occur in the receiving object. The behavior is accomplished when an operation is executed.

An operation within **SenderObject** generates a message of the form *message* (<parameters>) where the parameters identify **ReceiverObject** as the object to be

stimulated by the message, the operation within **ReceiverObject** that is to receive the

message, and the data items that provide information that is required for the operation

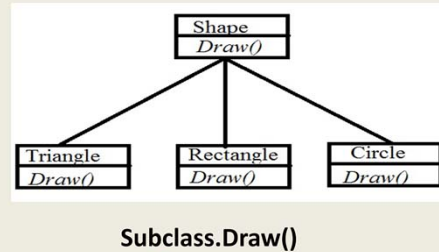
to be successful. The collaboration defined between classes as part of the requirements

model provides useful guidance in the design of messages.

Data/class design

- Polymorphism
 - reduces the effort required to extend the design of an existing object-oriented system

```
case of graphtype:  
  if graphtype = linegraph then DrawLineGraph (data);  
  if graphtype = piechart then DrawPieChart (data);  
  if graphtype = histogram then DrawHisto (data);  
  if graphtype = kiviak then DrawKiviak (data);  
end case;
```



Ideally, once data are collected for a particular type of graph, the graph should draw itself.

To accomplish this in a conventional application (and maintain module cohesion), it

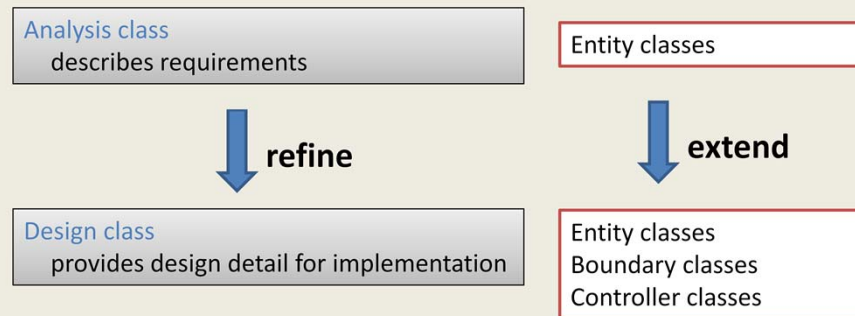
would be necessary to develop drawing modules for each type of graph.

in an object-oriented system, all of the graphs become subclasses of a general class called **Graph**. Using a concept called *overloading* [Tay90], each subclass defines an operation called *draw*. An object can send a *draw* message

to any one of the objects instantiated from any one of the subclasses

Data/class design

- Design classes



The analysis model defines a set of analysis classes.

The level of abstraction of an analysis class is relatively high

Design classes refine the analysis classes

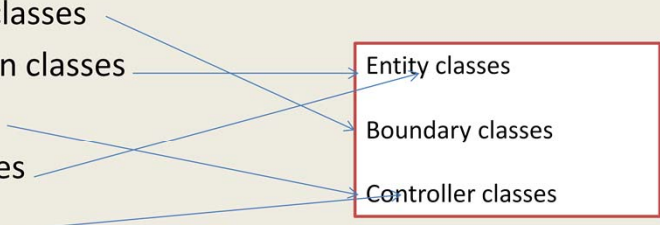
Design class provides design detail that will enable the classes to be implemented, and implement a software infrastructure that supports the business solution.

Controller classes are designed to manage (1) the creation or update of entity objects, (2) the instantiation of boundary objects as they obtain information from entity objects, (3) complex communication between sets of objects, and (4) validation

of data communicated between objects or between the user and the application.

Data/class design

- Design classes

- User interface classes
 - Business domain classes
 - Process classes
 - Persistent classes
 - System classes
- 
- The diagram shows five design classes on the left and three implementation classes on the right. Arrows indicate the following mappings: User interface classes to Boundary classes; Business domain classes to Entity classes; Process classes to Boundary classes; Persistent classes to Entity classes; and System classes to Controller classes. The implementation classes are enclosed in a red box.
- Entity classes
 - Boundary classes
 - Controller classes

This is a finer classification of design classes. (the mapping is my personal opinion!)

User interface classes define all abstractions that are necessary for human computer interaction (HCI).

Business domain classes are often refinements of the analysis classes defined earlier. The classes identify the attributes and services (methods) that are required to implement some element of the business domain.

Process classes implement lower-level business abstractions required to fully manage the business domain classes.

Persistent classes represent data stores (e.g., a database) that will persist beyond the execution of the software.

System classes implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

Data/class design

- Characteristics of well-formed design class
 - Complete and sufficient
 - Primitiveness
 - High cohesion
 - Low coupling

A design class should be the **complete encapsulation** of *all attributes and methods* that can reasonably be expected to exist for the class.

•E.g.) the class **Scene** defined for video-editing software is complete only if it contains all attributes and methods that can reasonably be associated with the creation of a video scene. **Sufficiency** ensures that the design class contains only those methods that are sufficient to achieve the intent of the class, no more and no less.

Methods associated with a design class should be focused on accomplishing **one service for the class**.

Once the service has been implemented with a method, the class should not provide another way to accomplish the same thing.

High cohesion.

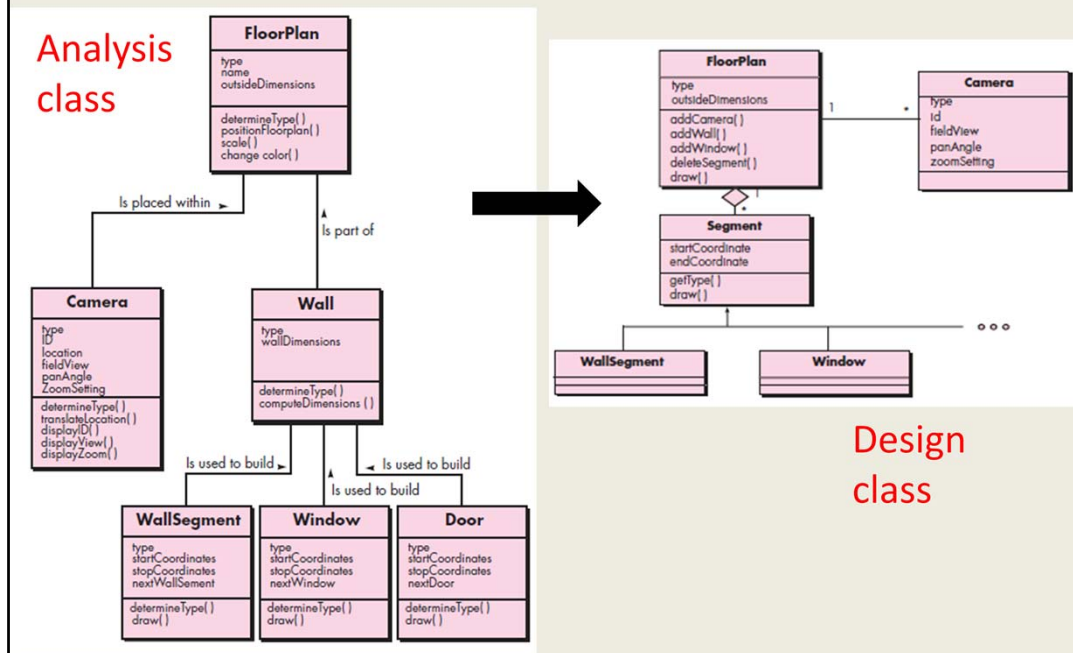
A cohesive design class has a small, focused set of responsibilities and single-mindedly applies attributes and methods to implement

•E.g.) The class **VideoClip** might contain a set of methods for editing the video clip. As long as each method focuses solely on attributes associated with the video clip, cohesion is maintained.

Low coupling.

Within the design model, it is necessary for design classes to collaborate with one another. However, collaboration should be kept to an acceptable minimum. In general, design classes within a subsystem should have only limited knowledge of other classes.

Data/class design



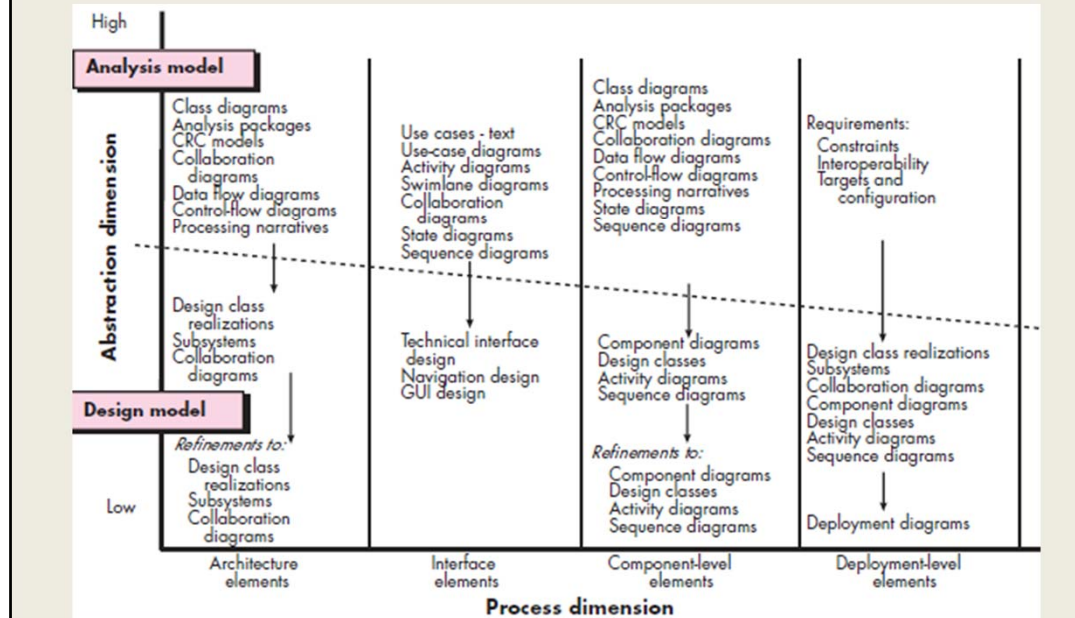
Example (SafeHome) - Design class for FloorPlan and composite aggregation for the class (see sidebar discussion)

The analysis class showed only things in the problem domain, well, actually on the computer screen, that were visible to the end user, right?

Ed: Yep, but for the **FloorPlan** design class, I've got to add some things that are implementation specific. I needed to show that **FloorPlan** is an aggregation of segments—hence the **Segment** class—and that the **Segment** class is composed of lists for wall segments, windows, doors, and so on. The class **Camera** collaborates with **FloorPlan**, and obviously, there can be many cameras in the floor plan.

Design model

- Two dimensions



Now we already there are two kinds of models to build during the modeling activity;

The design model can be viewed in two different dimensions

The *process dimension* indicates the evolution of the design model as design tasks are executed as part of the software process

The *abstraction dimension* represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively.

model elements indicated along the horizontal axis are not always developed in a sequential fashion. In most cases preliminary architectural

design sets the stage and is followed by interface design and component-level design, which often occur in parallel. The deployment model is usually delayed until the design has been fully developed.

Design model

- versus analysis model

The elements of the design model use many of **the same UML diagrams** that were used in the analysis model.

the difference is that these diagrams are **refined and elaborated** as part of design

The difference is that these diagrams are refined and elaborated as part of design; more implementation-specific detail is provided, and architectural structure and style, components that reside within the architecture, and interfaces between the components and with the outside world are all emphasized.

Design model

- Model Elements
 - Data elements
 - Architectural elements
 - Interface elements
 - Component elements
 - Deployment elements

Design model – data elements

- Data design / data architecting
 - Data model ->
 - Data structures [component level]
 - Database/file architecture [architectural level]

Data design (sometimes referred to as *data architecting*) creates a model of data and/or information that is represented at a high level of abstraction (the customer/user's view of data).

This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system.

At the architectural level, data design focuses on files or databases

At the component level, data design considers the data structures that are required to implement local data objects

Design model – architectural elements

- Architectural modeling
 - Application domain
 - Analysis classes, their relationships, collaborations and behaviors are transformed into design realizations
 - Patterns and “styles”

The *architectural design* for software is the equivalent to the floor plan of a house.

The floor plan depicts the overall layout of the rooms; their size, shape, and relationship to one another; and the doors and windows that allow movement into and out of the rooms. The floor plan gives us an overall view of the house.

Architectural design elements give us an overall view of the software.

The architectural model [Sha96] is derived from three sources:

- 1) information about the application domain for the software to be built;
- 2) specific requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand, and
- 3) the availability of architectural patterns (Chapter 12) and styles(Chapter 9).

Design model – interface elements

- Interface modeling
 - the **user interface** (UI)
 - **external interfaces** to other systems, devices, networks or other producers or consumers of information
 - **internal interfaces** between various design components.

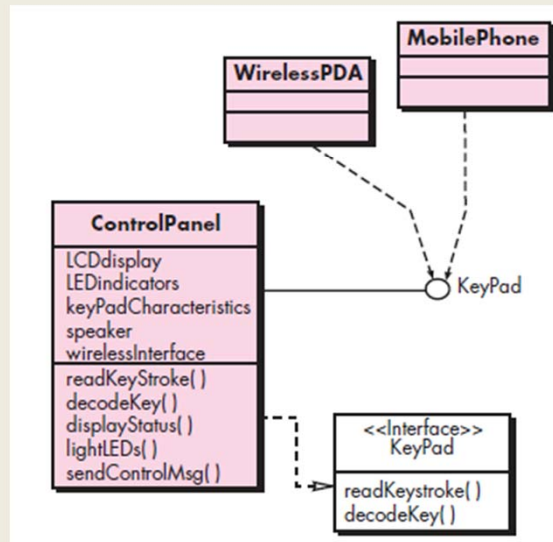
Interface is a set of operations that describes the externally observable behavior of a class and provides access to its public operations

Modeled using UML communication/collaboration diagrams

an interface is a set of operations that describes some part of the behavior of a class and provides access to these operations.

Design model – interface elements

- Interface modeling



In some cases, an interface is modeled in much the same way as a class;
What other diagram does this diagram look like? (class diagram)

Understand the relationships between the single diagrams [dependence, realization]: wirelessPDA and MobilePhone depend on ControlPanel (as they both use the KeyPad function of ControlPanel), using the circle to show the dependency is reflected by 'using the KeyPad'

Design model – component-level elements

- Component modeling
 - Internal details of each component
 - [Data structures](#) for all local data objects
 - [Algorithmic detail](#) for all component processing that occurs within a component
 - [Interface](#) that allows access to all component operations

The component-level design for software is the equivalent to a set of detailed drawings (and specifications) for each room in a house.

Fully describes the internal detail of each software component;

Modeled using UML component diagrams, UML activity diagrams, pseudocode (PDL), and sometimes flowcharts

Design model – component-level elements

- Component modeling with [component diagram](#)

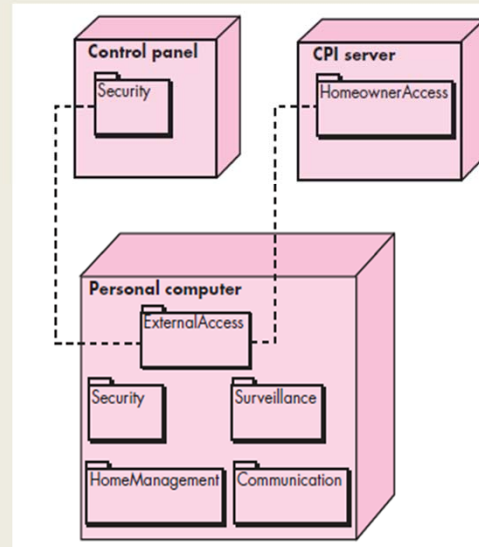


In this figure, a component named **SensorManagement** (part of the *SafeHome* security function) is represented. A dashed arrow connects the component to a class named **Sensor** that is assigned to it.

The **SensorManagement** component performs all functions associated with *SafeHome* sensors including monitoring and configuring them.

Design model – deployment-level elements

- Deployment modeling
 - Descriptor form deployment
 - Instance form deployment



Indicates how software functionality and subsystems will be allocated within the physical computing environment

Modeled using UML deployment diagrams

Descriptor form deployment diagrams show the computing environment but does not indicate configuration details

Instance form deployment diagrams identifying specific named hardware configurations are developed during the latter stages of design

This figure is a descriptor form deployment diagram (*SafeHome* product are configured to operate within three primary computing environments—a home-based PC, the *SafeHome* control panel, and a server housed **at CPI Corp.**

three computing environments are shown, The subsystems (functionality) housed within each computing element are indicated

the personal computer houses subsystems that implement security, surveillance, home management, and communications features. In addition, an external access subsystem has been designed to manage all attempts to access the

SafeHome system from an external source.

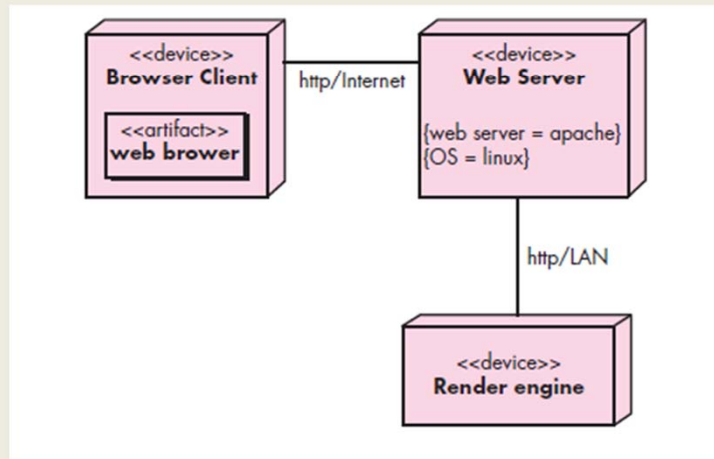
This form of deployment does not explicitly indicate configuration details. For example, the “personal computer” is not further identified.

It could be a Mac or a Windows-based PC, a Sun workstation, or a Linux-box. These

details are provided when the deployment diagram is revisited in *instance form* during the latter stages of design or as construction begins.

Design model – deployment-level elements

- Deployment modeling with **deployment diagram**



useful for showing the physical distribution of a software system among hardware platforms and execution environments.

three hardware devices involved in your system: the Web client (the users' computer running a browser), the computer hosting the Web server, and the computer hosting the rendering engine.

hardware components are drawn in boxes labeled with “<<device>>”.

Communication paths between hardware components are drawn with lines with optional labels (e.g., showing communication protocol and the type of network used to connect the devices)

Each node in a deployment diagram can also be annotated with details about the device.

Deployment diagrams can also display execution environment nodes, which are drawn as boxes containing the label “<<execution environment>>”. These nodes represent systems, such as operating systems, that can host other software.

Summary

- OO design concepts
- Data/class design
 - In relation to analysis classes
- Design model
 - Dimensions
 - Model elements