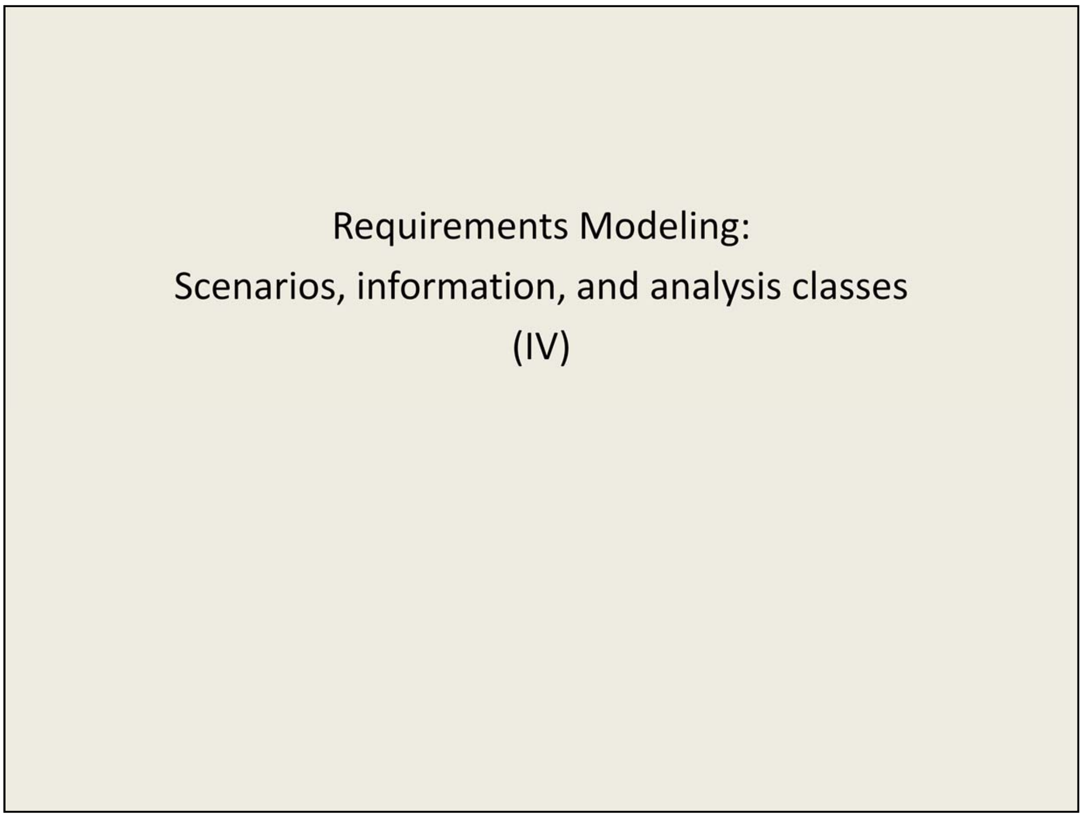
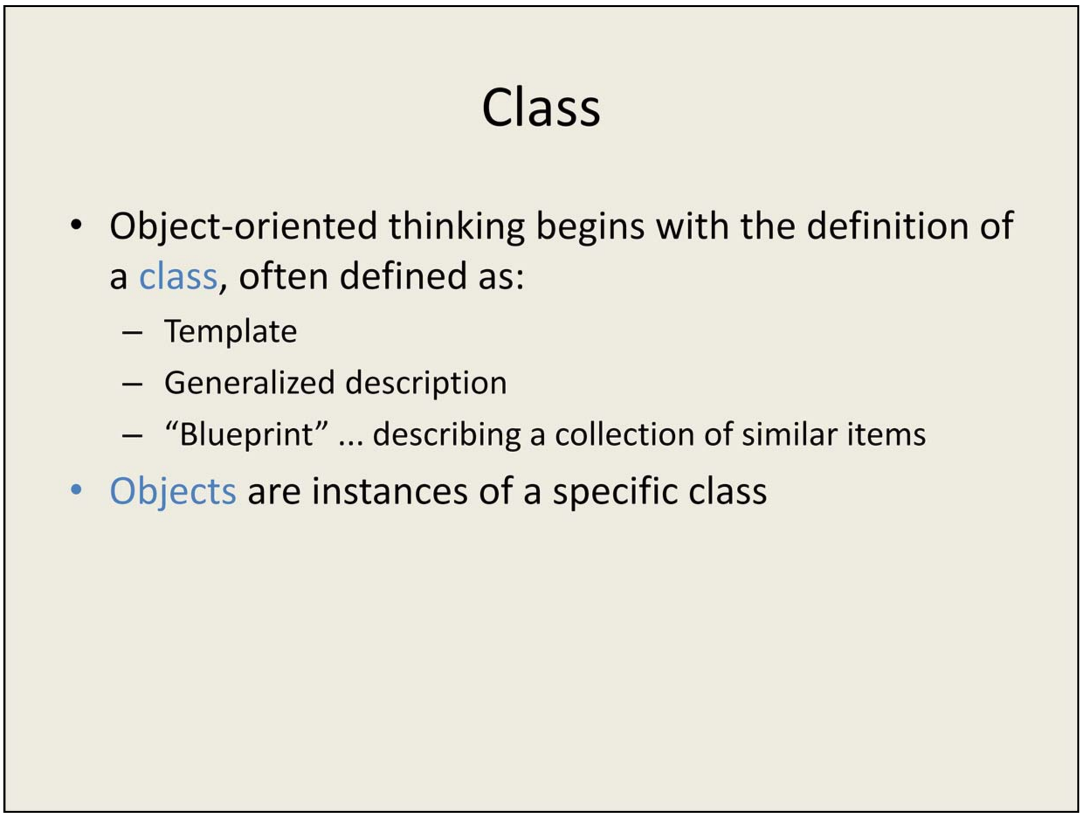


Requirements Modeling:
Scenarios, information, and analysis classes
(IV)

Object-Oriented Concepts

- Underlying OO analysis and design
  - OO: objects + classification + inheritance + communication
- Key concepts:
  - Classes and objects
  - Attributes and operations
  - Encapsulation and instantiation
  - Inheritance

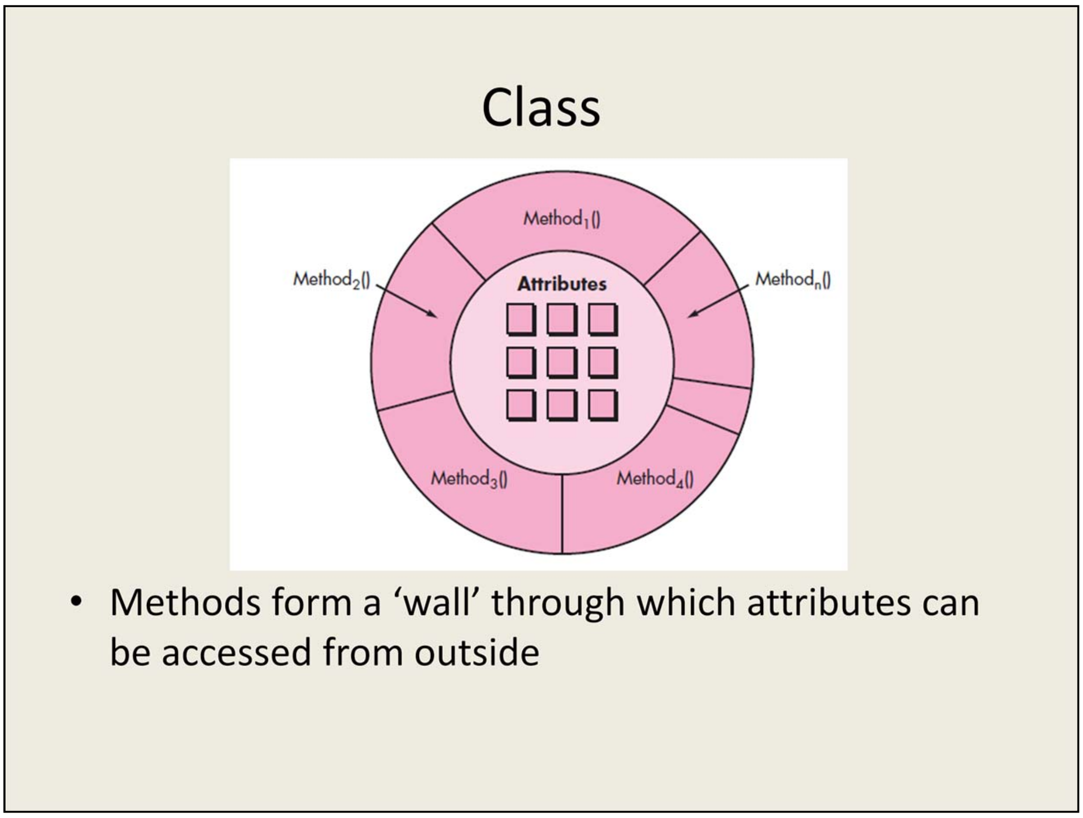


Class

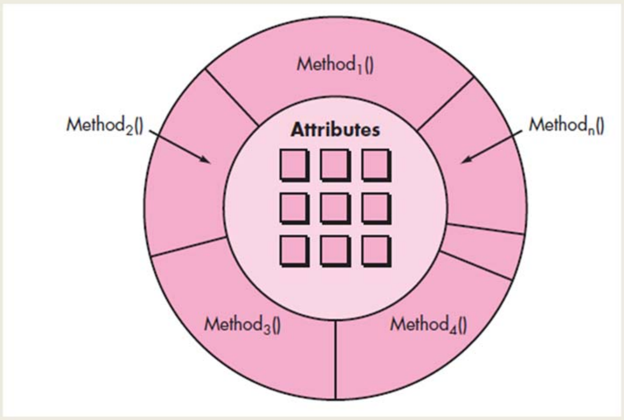
- Object-oriented thinking begins with the definition of a class, often defined as:
  - Template
  - Generalized description
  - “Blueprint” ... describing a collection of similar items
- Objects are instances of a specific class

By definition, objects are instances of a specific class and inherit its attributes and the operations that are available to manipulate the attributes.

A class is an OO concept that encapsulates the data and procedural abstractions required to describe the content and behavior of some real-world entity.

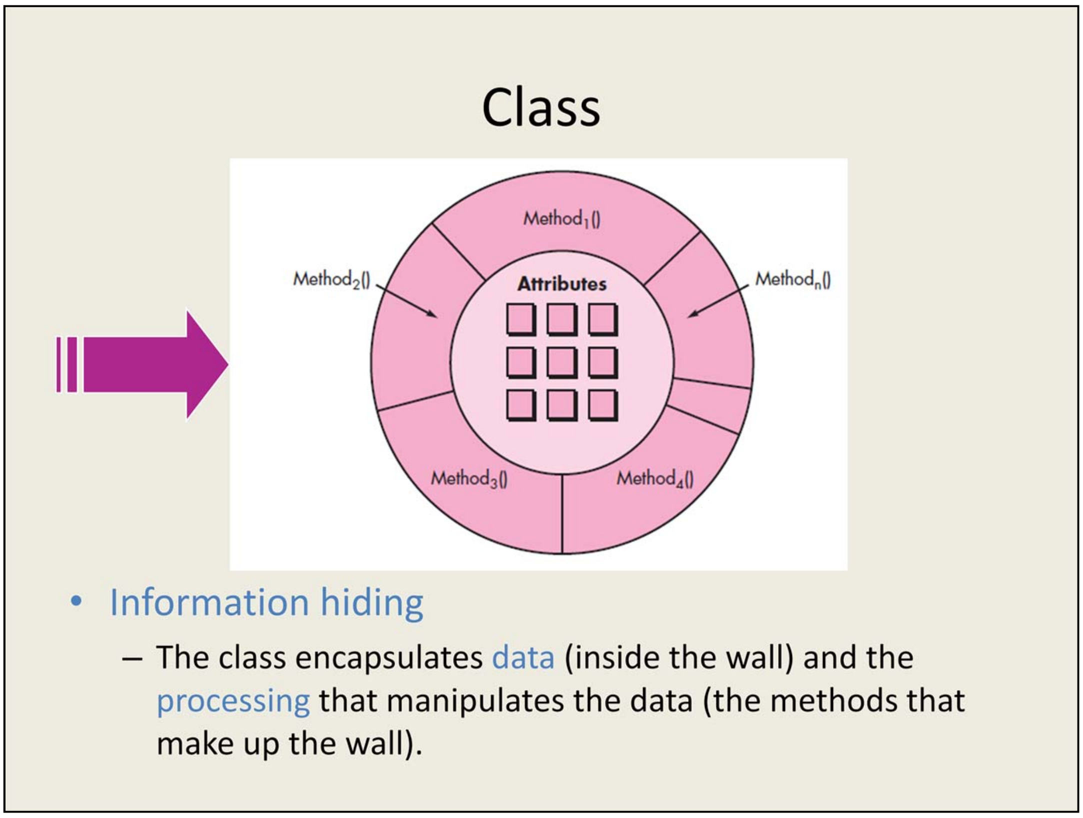


Class

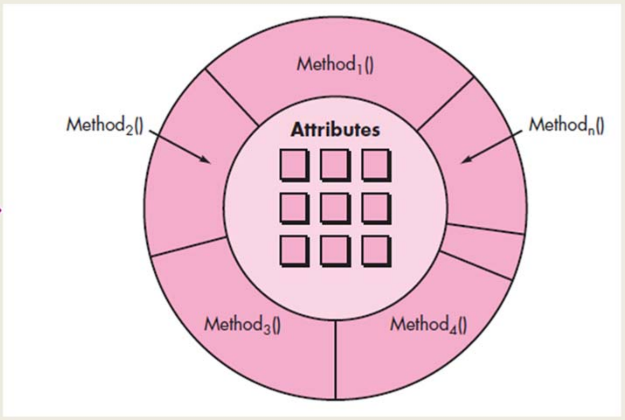


- Methods form a ‘wall’ through which attributes can be accessed from outside

In a well-designed class, the only way to reach the attributes (and operate on them) is to go through one of the methods that form the “wall” illustrated in the figure.

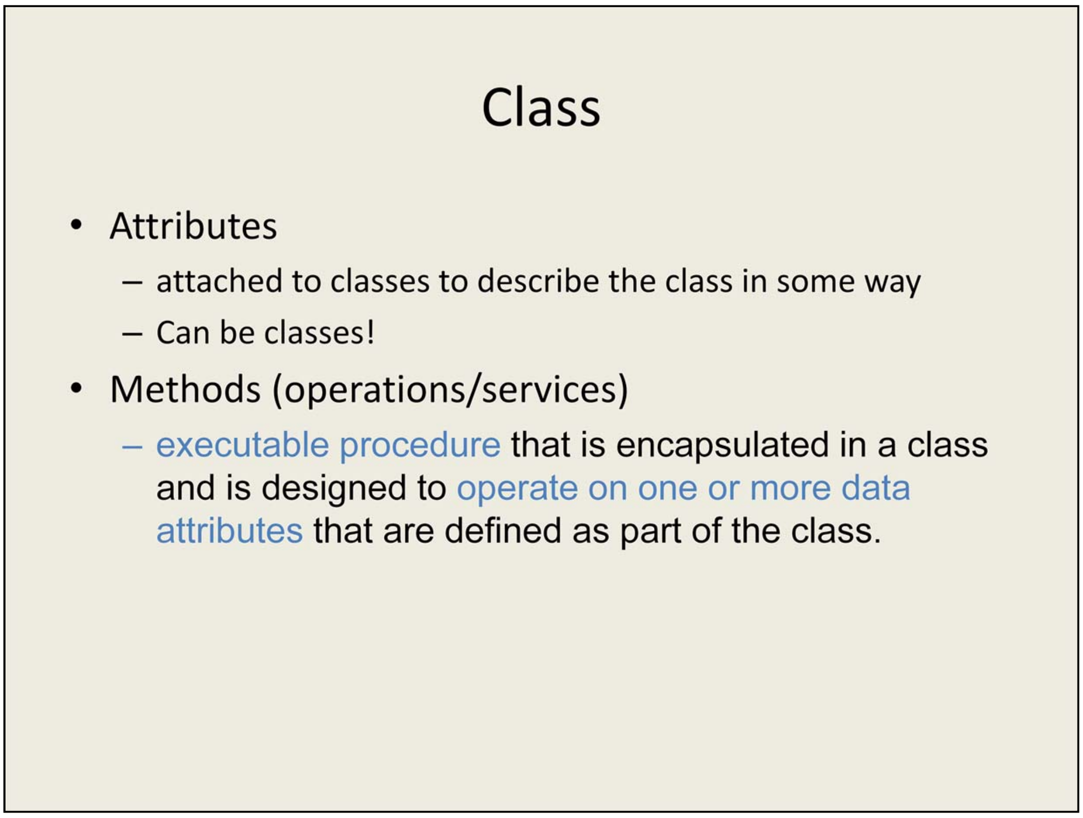


Class



- Information hiding
  - The class encapsulates data (inside the wall) and the processing that manipulates the data (the methods that make up the wall).

This achieves information hiding (Chapter 8) and reduces the impact of side effects associated with change. Since the methods tend to manipulate a limited number of attributes, their **cohesion** is improved, and because communication occurs only through the methods that make up the "wall," the class tends to be **less strongly coupled** from other elements of a system.1



Class

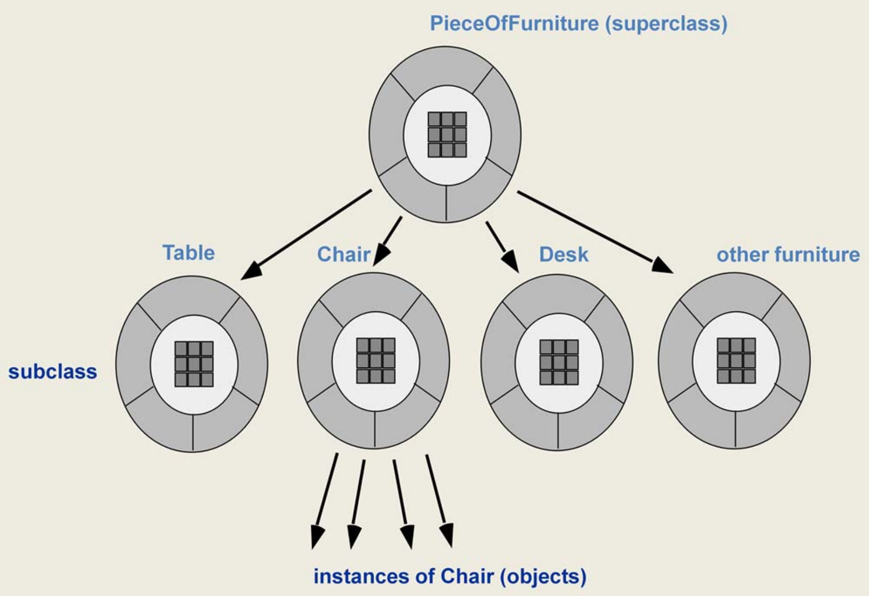
- Attributes
  - attached to classes to describe the class in some way
  - Can be classes!
- Methods (operations/services)
  - executable procedure that is encapsulated in a class and is designed to operate on one or more data attributes that are defined as part of the class.

An object encapsulates data (represented as a collection of attributes) and the **algorithms** that process the data. These algorithms are called *operations, methods,* or *services*2 and can be viewed as processing components.

Class hierarchy

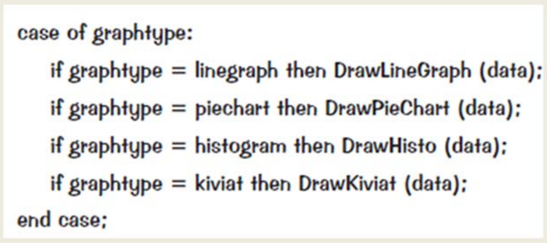
- A superclass (or base class) a generalization of a set of classes that are related to it
- A subclass is a specialization of the superclass
- Inheritance
  - A subclass inherits all of the attributes and operations associated with its superclass

Class hierarchy

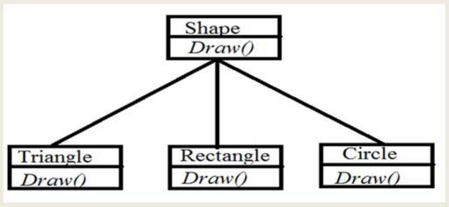


Class hierarchy

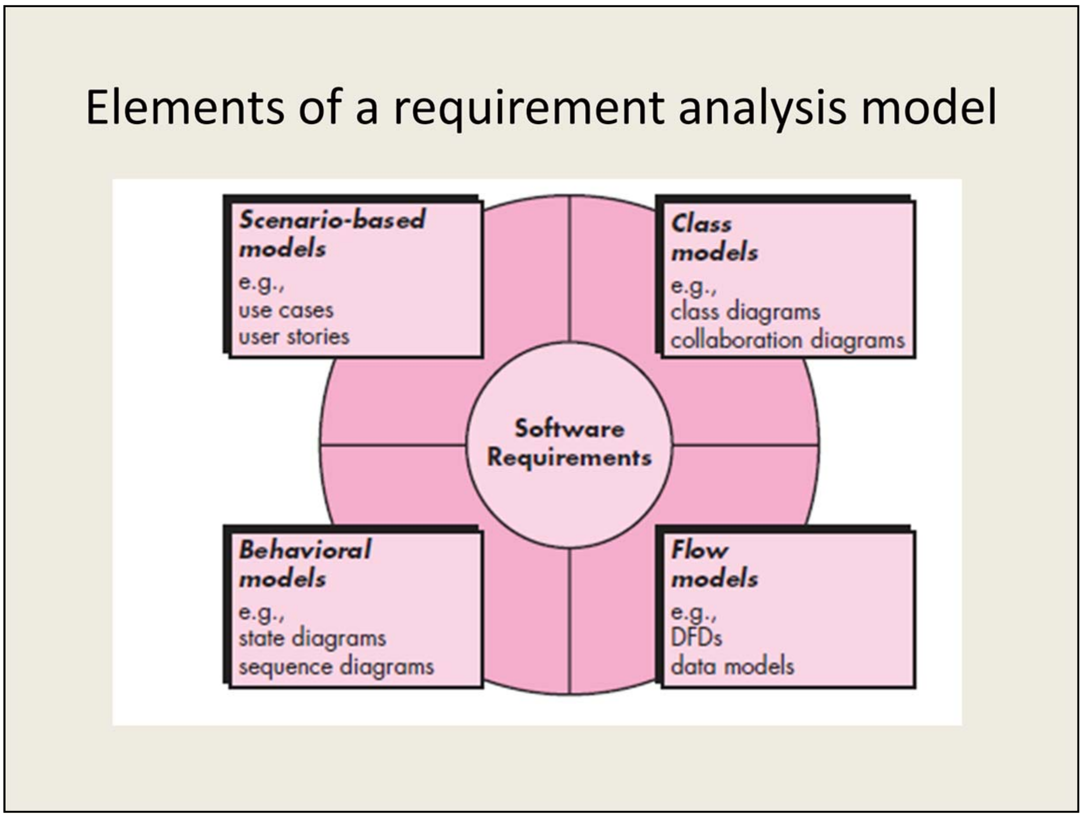
- Polymorphism
  - reduces the effort required to extend the design of an existing object-oriented system



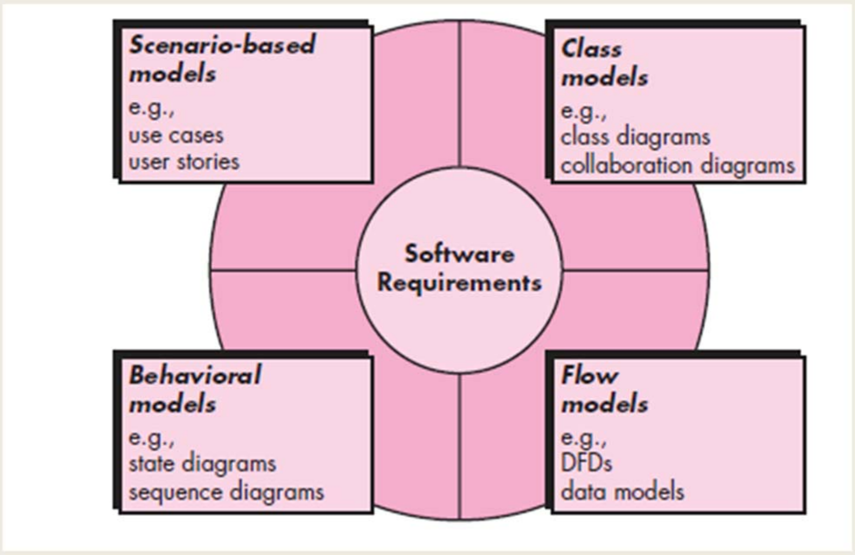
```
case of graphtype:
    if graphtype = linegraph then DrawLineGraph (data);
    if graphtype = piechart then DrawPieChart (data);
    if graphtype = histogram then DrawHisto (data);
    if graphtype = kiviat then DrawKiviat (data);
end case;
```



Subclass.Draw()



Elements of a requirement analysis model

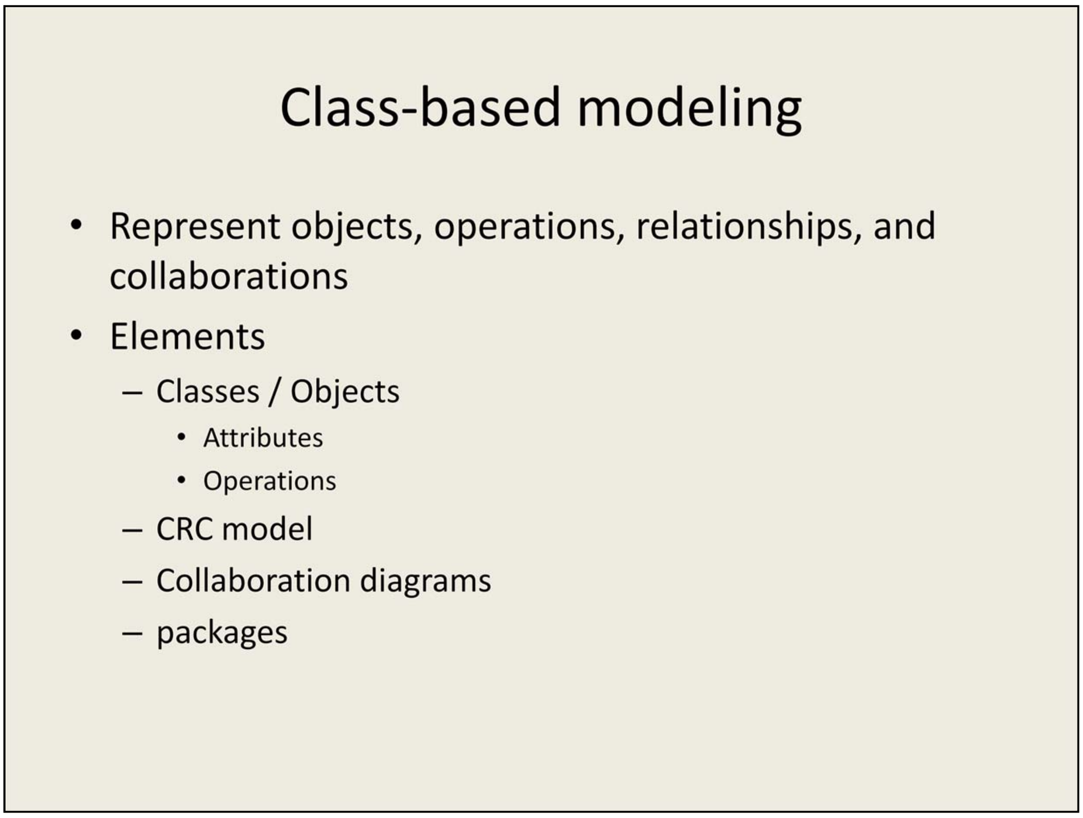

Scenario-based models
e.g., use cases user stories
Class models
e.g., class diagrams collaboration diagrams
Software Requirements
Behavioral models
e.g., state diagrams sequence diagrams
Flow models
e.g., DFDs data models

Each element of the requirements model (Figure 6.3) presents the problem from a different point of view. Scenario-based elements depict how the user interacts with the system and the specific sequence of activities that occur as the software is used.

Class-based elements model the objects that the system will manipulate, the operations that will be applied to the objects to effect the manipulation, relationships (some hierarchical) between the objects, and the collaborations that occur between the classes that are defined.

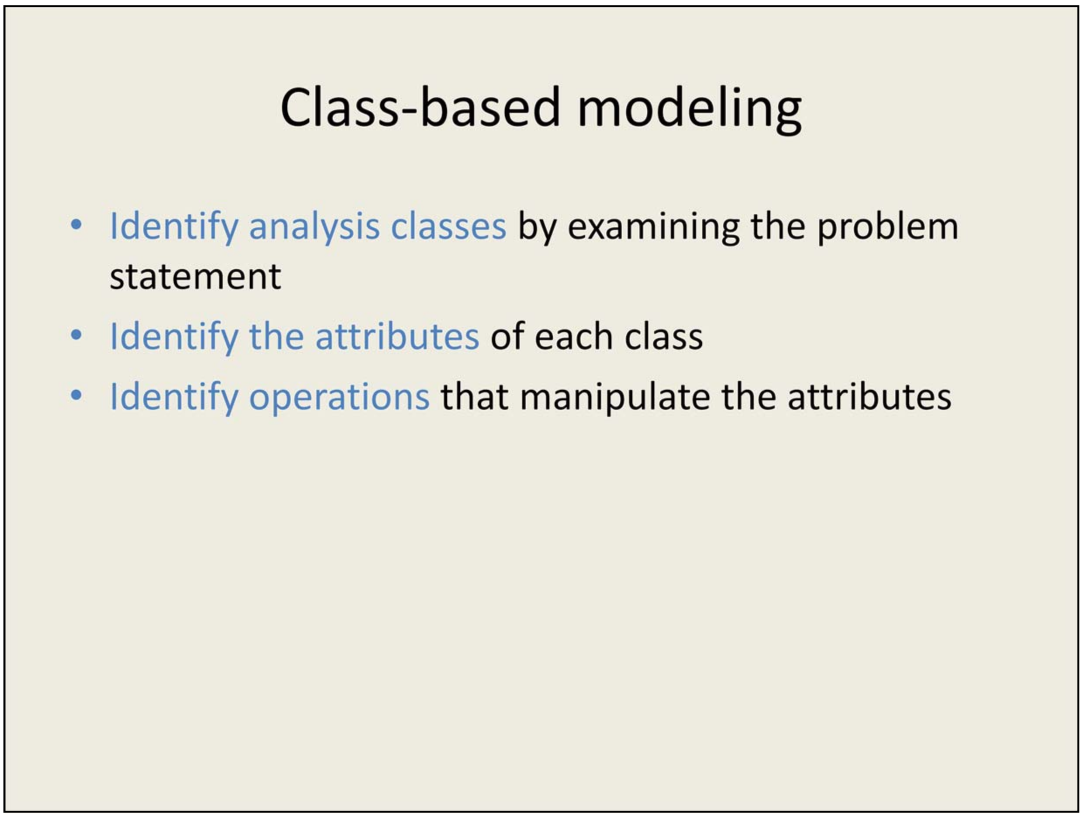
Behavioral elements depict how external events change the state of the system or the classes that reside within it.

Finally, flow-oriented elements represent the system as an information transform, depicting how data objects are transformed as they flow through various system functions.



Class-based modeling

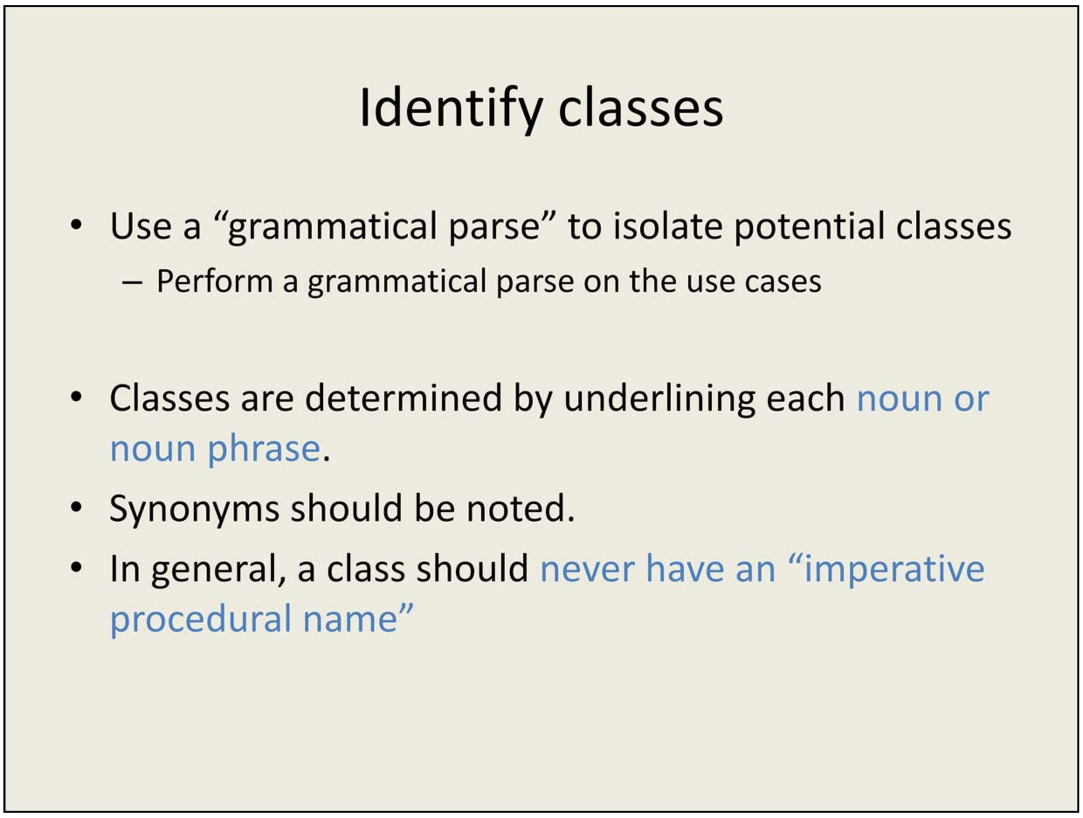
- Represent objects, operations, relationships, and collaborations
- Elements
  - Classes / Objects
    - Attributes
    - Operations
  - CRC model
  - Collaboration diagrams
  - packages



Class-based modeling

- Identify analysis classes by examining the problem statement
- Identify the attributes of each class
- Identify operations that manipulate the attributes

Core tasks in class-based modeling



Identify classes

- Use a "grammatical parse" to isolate potential classes
  - Perform a grammatical parse on the use cases

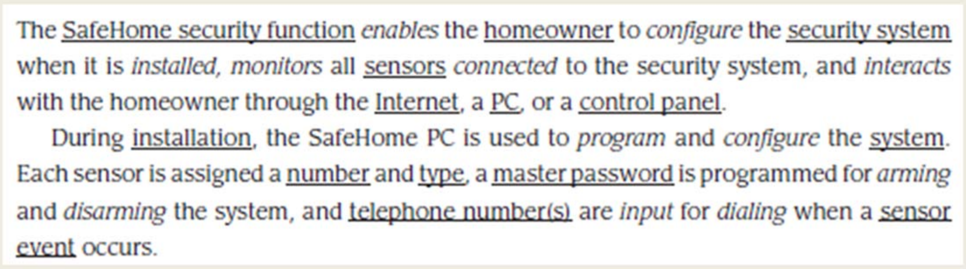
- Classes are determined by underlining each noun or noun phrase.
- Synonyms should be noted.
- In general, a class should never have an "imperative procedural name"

If the class (noun) is required to *implement a solution*, then it is part of the solution space; otherwise, if a class is necessary *only to describe a solution*, it is part of the problem space.

Analysis classes (basic types)

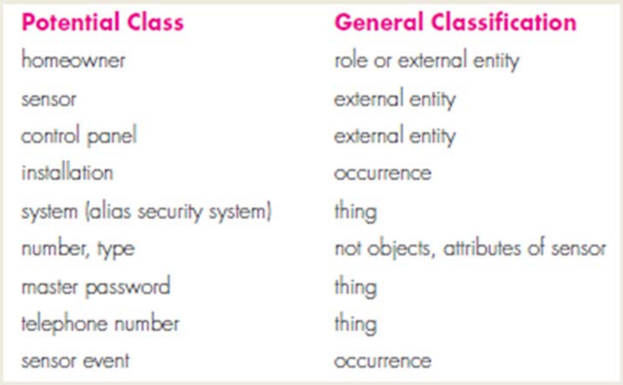
- External entities (e.g., other systems, devices, people) that produce or consume information to be used by a computer-based system.
- Things (e.g, reports, displays, letters, signals) that are part of the information domain for the problem.
- Occurrences or events (e.g., a property transfer or the completion of a series of robot movements) that occur within the context of system operation.
- Roles (e.g., manager, engineer, salesperson) played by people who interact with the system.
- Organizational units (e.g., division, group, team) that are relevant to an application.
- Places (e.g., manufacturing floor or loading dock) that establish the context of the problem and the overall function of the system.
- Structures (e.g., sensors, four-wheeled vehicles, or computers) that define a class of objects or related classes of objects.

Example: SafeHome (security)

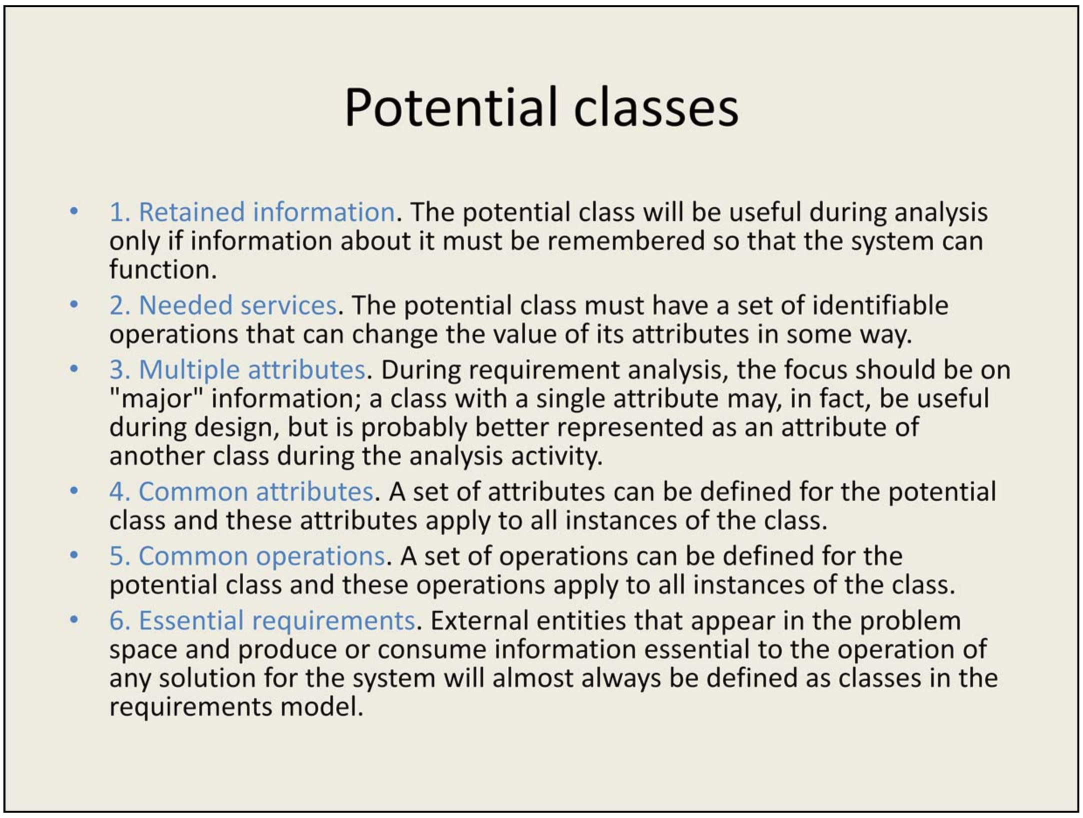


The SafeHome security function *enables* the homeowner to *configure* the security system when it is *installed, monitors* all sensors *connected* to the security system, and *interacts* with the homeowner through the Internet, a PC, or a control panel.

During installation, the SafeHome PC is used to *program* and *configure* the system. Each sensor is assigned a number and type, a master password is programmed for *arming* and *disarming* the system, and telephone number(s) are *input* for *dialing* when a sensor event occurs.

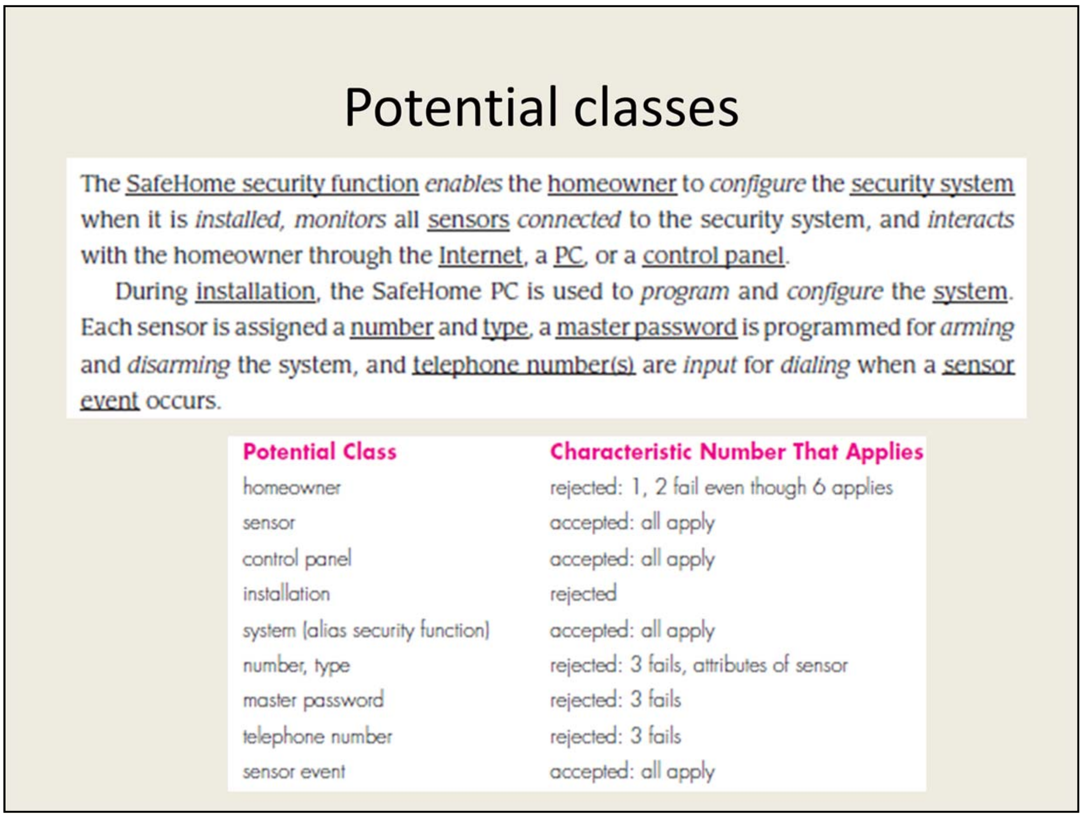


Potential Class	**General Classification**
homeowner	role or external entity
sensor	external entity
control panel	external entity
installation	occurrence
system (alias security system)	thing
number, type	not objects, attributes of sensor
master password	thing
telephone number	thing
sensor event	occurrence


Potential classes

- 1. Retained information. The potential class will be useful during analysis only if information about it must be remembered so that the system can function.
- 2. Needed services. The potential class must have a set of identifiable operations that can change the value of its attributes in some way.
- 3. Multiple attributes. During requirement analysis, the focus should be on "major" information; a class with a single attribute may, in fact, be useful during design, but is probably better represented as an attribute of another class during the analysis activity.
- 4. Common attributes. A set of attributes can be defined for the potential class and these attributes apply to all instances of the class.
- 5. Common operations. A set of operations can be defined for the potential class and these operations apply to all instances of the class.
- 6. Essential requirements. External entities that appear in the problem space and produce or consume information essential to the operation of any solution for the system will almost always be defined as classes in the requirements model.

To be considered a legitimate class for inclusion in the requirements model, a potential object should satisfy all (or almost all) of these characteristics.



Potential classes

The SafeHome security function *enables* the homeowner to *configure* the security system when it is *installed, monitors* all sensors *connected* to the security system, and *interacts* with the homeowner through the Internet, a PC, or a control panel.

During installation, the SafeHome PC is used to *program* and *configure* the system. Each sensor is assigned a number and type, a master password is programmed for *arming* and *disarming* the system, and telephone number(s) are *input* for *dialing* when a sensor event occurs.

Potential Class	Characteristic Number That Applies
homeowner	rejected: 1, 2 fail even though 6 applies
sensor	accepted: all apply
control panel	accepted: all apply
installation	rejected
system (alias security function)	accepted: all apply
number, type	rejected: 3 fails, attributes of sensor
master password	rejected: 3 fails
telephone number	rejected: 3 fails
sensor event	accepted: all apply

(1) the preceding list is not all-inclusive, additional classes would have to be added to complete the model;

(2) some of the rejected potential classes will become attributes for those classes that were accepted (e.g., number and

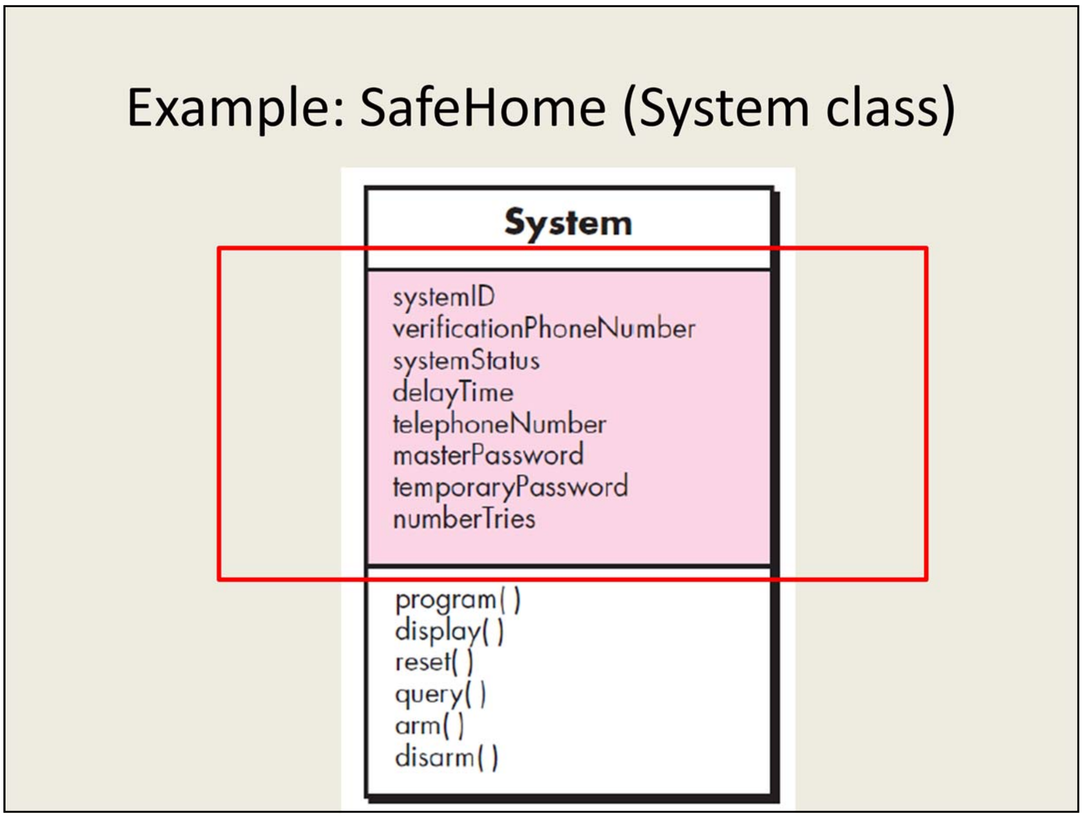
type are attributes of **Sensor,** and master password and telephone number may become attributes of **System**);

(3) different statements of the problem might cause different “accept or reject” decisions to be made (e.g., if each homeowner had an individual

password or was identified by voice print, the **Homeowner** class would satisfy characteristics 1 and 2 and would have been accepted).

Identify attributes

- Attributes describe a class that has been selected for inclusion in the analysis model.
- Define the class in the context of problem space
  - build two different classes for professional baseball players
    - For Playing Statistics software: name, position, batting average, fielding percentage, years played, and games played might be relevant
    - For Pension Fund software: average salary, credit toward full vesting, pension plan options chosen, mailing address, and the like.



Example: SafeHome (System class)

Sensors are part of the overall *SafeHome* system, and yet they are not listed as

data items or as attributes in Figure 6.9. **Sensor** has already been defined as a class,

and multiple **Sensor** objects will be associated with the **System** class.

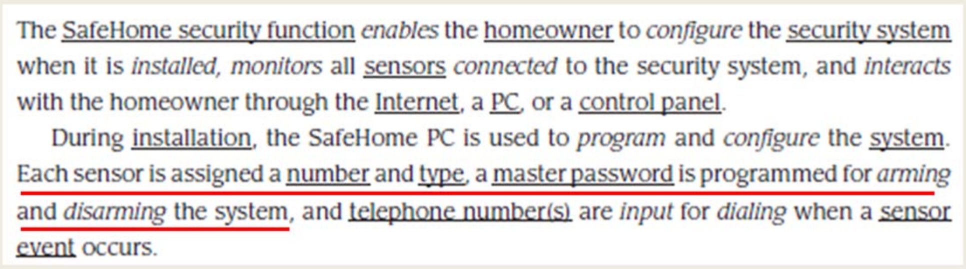
In general, we avoid defining an item as an attribute if more than one of the items is to be associated

with the class.

Identify operations

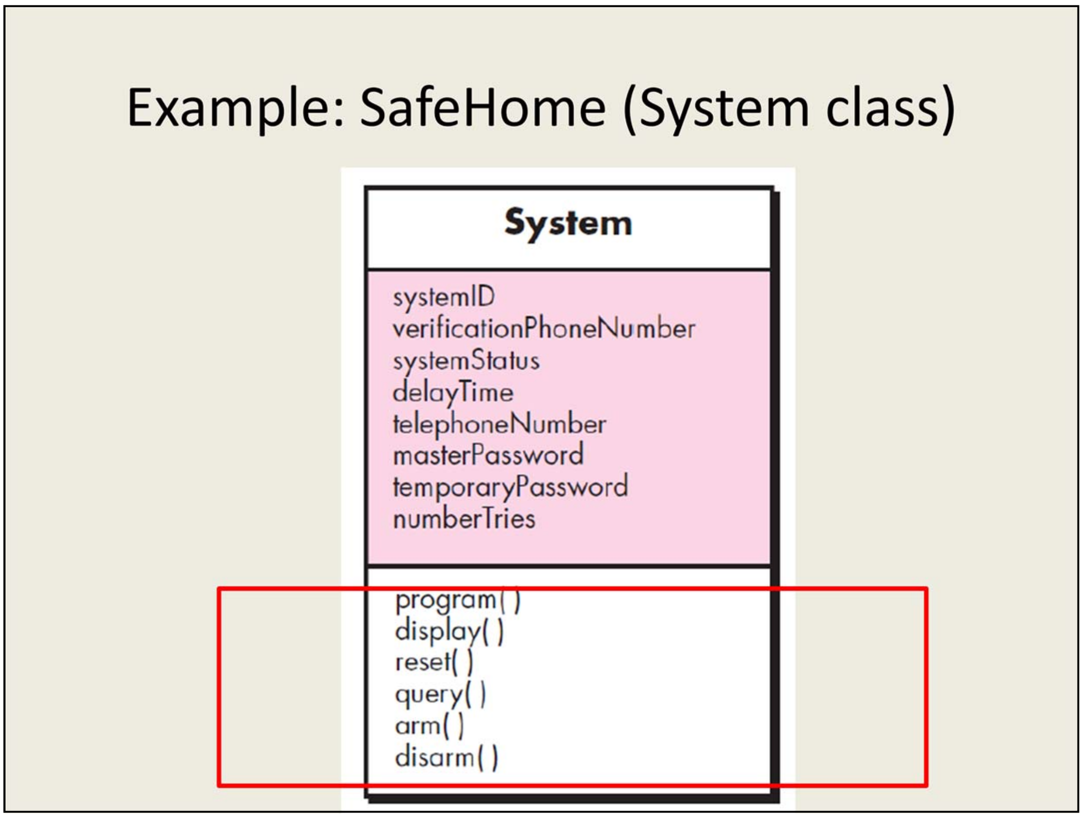
- Do a grammatical parse of a processing narrative and look at the verbs
- Operations can be divided into four broad categories:
  - (1) operations that manipulate data in some way (e.g., adding, deleting, reformatting, selecting)
  - (2) operations that perform a computation
  - (3) operations that inquire about the state of an object, and
  - (4) operations that monitor an object for the occurrence of a controlling event.

Example: SafeHome


The SafeHome security function *enables* the homeowner to *configure* the security system when it is *installed, monitors* all sensors *connected* to the security system, and *interacts* with the homeowner through the Internet, a PC, or a control panel.

During installation, the SafeHome PC is used to *program* and *configure* the system. Each sensor is assigned a number and type, a master password is programmed for *arming* and *disarming* the system, and telephone number(s) are *input* for *dialing* when a sensor event occurs.

- That an *assign()* operation is relevant for the **Sensor** class.
- That a *program()* operation will be applied to the **System** class.
- That *arm()* and *disarm()* are operations that apply to **System** class.

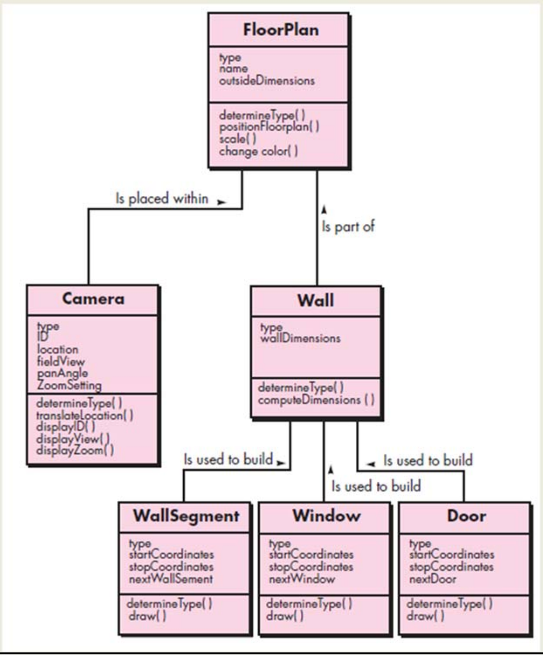


Example: SafeHome (System class)

In addition to the grammatical parse, you can gain additional insight into other operations by *considering the communication that occurs between objects*.

Objects communicate by passing messages to one another.

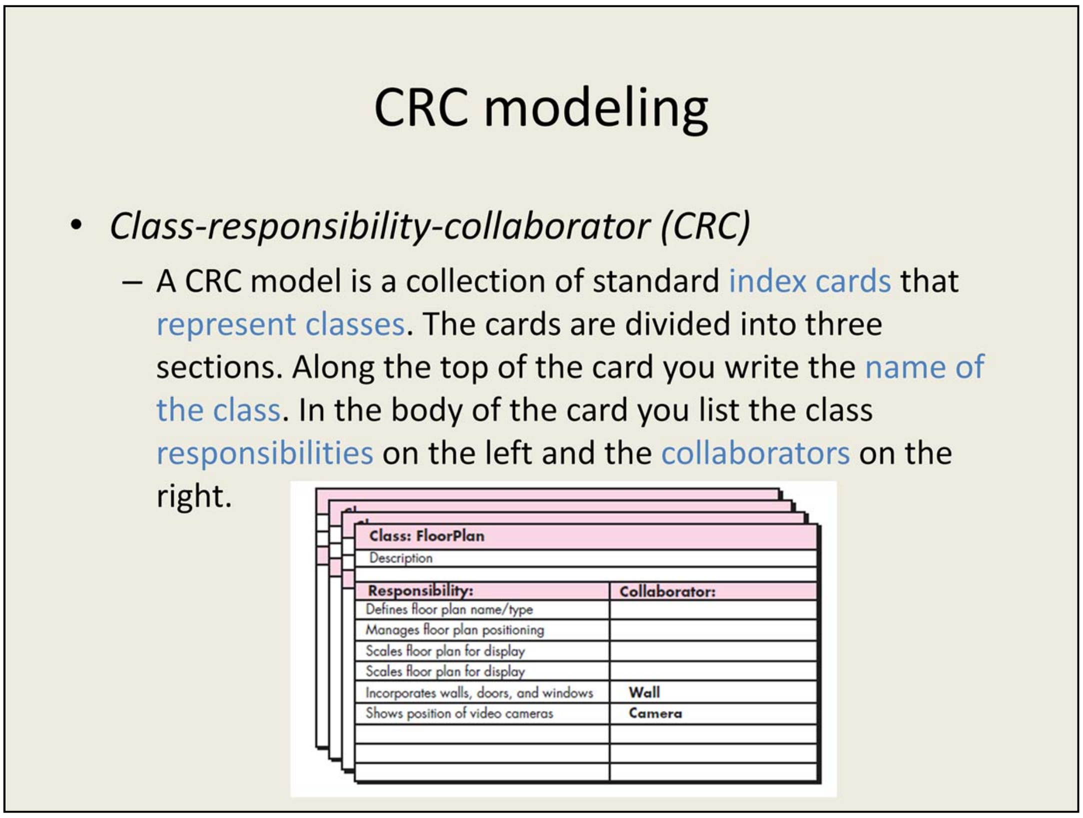
Example: SafeHome (FloorPlan class)



In the SafeHome, ACS-DCV use case. In this diagram, shown is not only classes but relationships between classes.

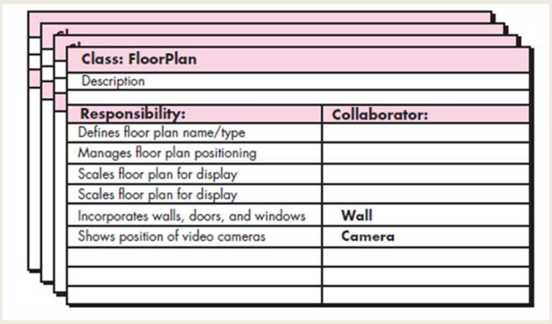
In addition, there are also the communication between classes. This communication is realized through passing messages

Between related classes. This communication can be modeled using the CRC model.



CRC modeling

- *Class-responsibility-collaborator (CRC)*
  - A CRC model is a collection of standard index cards that represent classes. The cards are divided into three sections. Along the top of the card you write the name of the class. In the body of the card you list the class responsibilities on the left and the collaborators on the right.

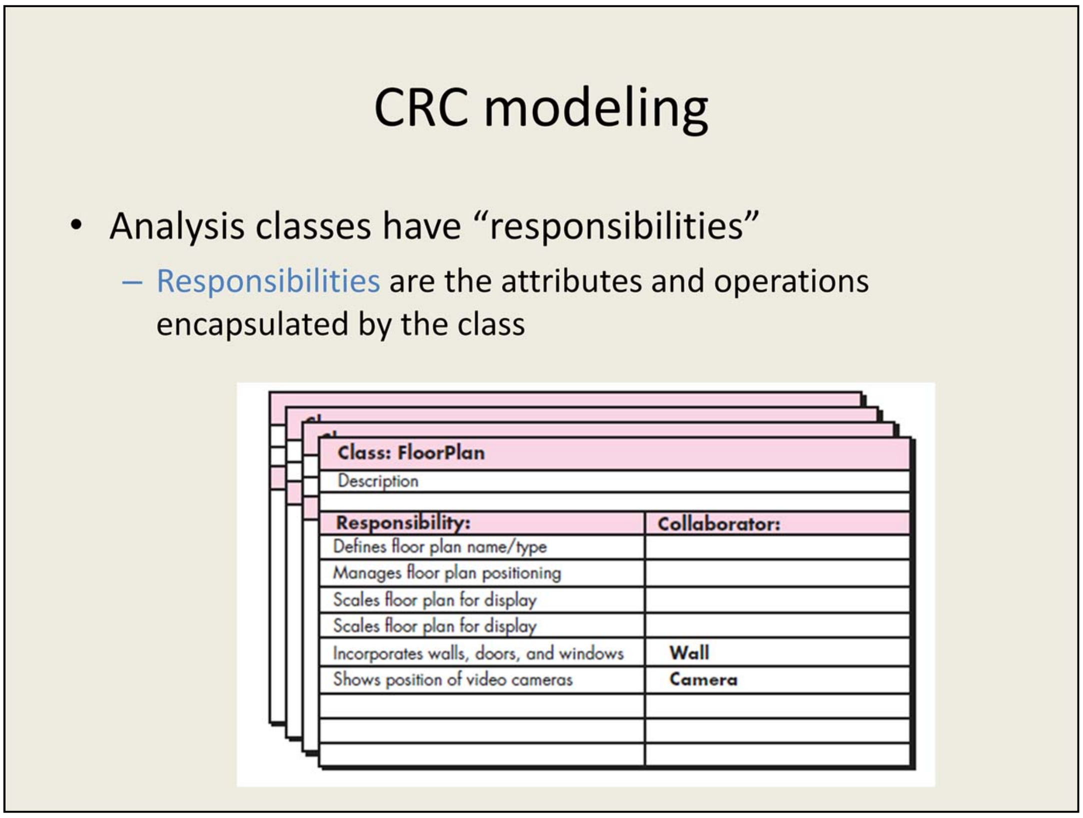


Class: FloorPlan

Description

Responsibility:	**Collaborator:**
Defines floor plan name/type	
Manages floor plan positioning	
Scales floor plan for display	
Scales floor plan for display	
Incorporates walls, doors, and windows	**Wall**
Shows position of video cameras	**Camera**

A simple CRC index card for the **FloorPlan** class is illustrated



CRC modeling

- Analysis classes have “responsibilities”
  - Responsibilities are the attributes and operations encapsulated by the class

Class: FloorPlan

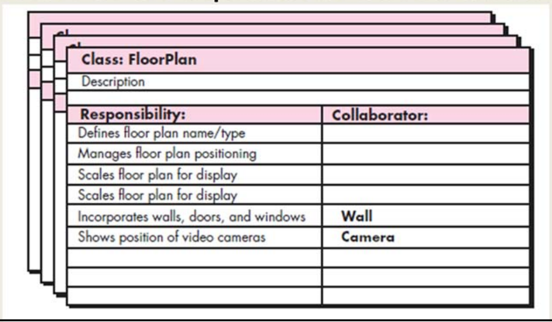
Description

Responsibility:	**Collaborator:**
Defines floor plan name/type	
Manages floor plan positioning	
Scales floor plan for display	
Scales floor plan for display	
Incorporates walls, doors, and windows	**Wall**
Shows position of video cameras	**Camera**

Responsibilities are the attributes and operations that are relevant for the class. Stated simply, a responsibility is “anything the class knows or does”

CRC modeling

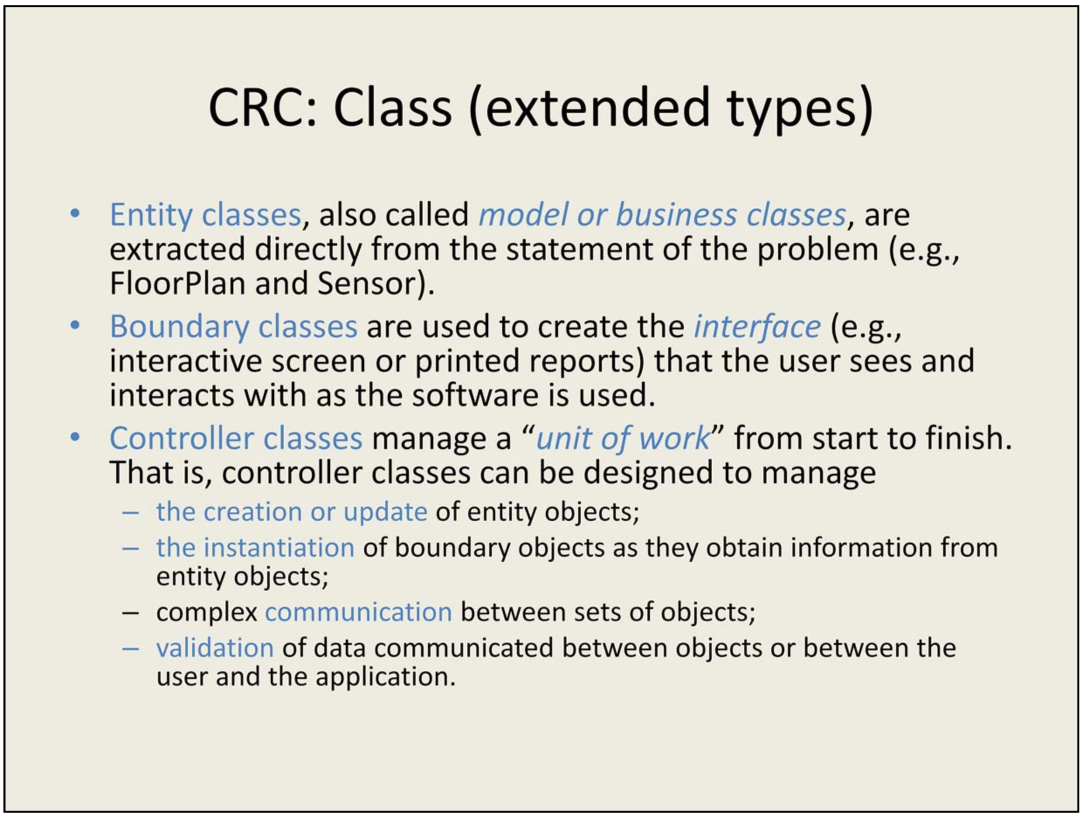
- Analysis classes collaborate with one another
  - Collaborators are those classes that are required to provide a class with the information needed to complete a responsibility.
- In general, a collaboration implies either a request for information or a request for some action.



Class: FloorPlan

Description

Responsibility:	Collaborator:
Defines floor plan name/type	
Manages floor plan positioning	
Scales floor plan for display	
Scales floor plan for display	
Incorporates walls, doors, and windows	**Wall**
Shows position of video cameras	**Camera**



CRC: Class (extended types)

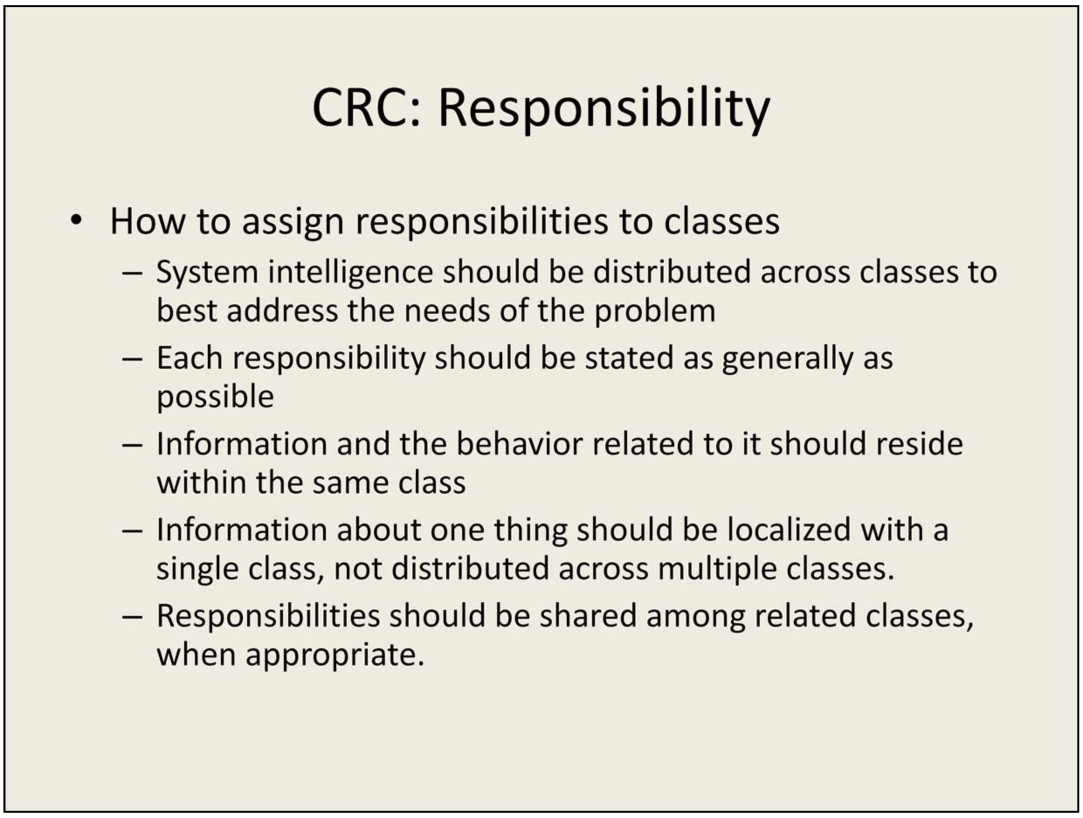
- Entity classes, also called *model or business classes*, are extracted directly from the statement of the problem (e.g., FloorPlan and Sensor).
- Boundary classes are used to create the *interface* (e.g., interactive screen or printed reports) that the user sees and interacts with as the software is used.
- Controller classes manage a "*unit of work*" from start to finish. That is, controller classes can be designed to manage
  - the creation or update of entity objects;
  - the instantiation of boundary objects as they obtain information from entity objects;
  - complex communication between sets of objects;
  - validation of data communicated between objects or between the user and the application.

See slide 14 for basic types of classes.

Entity class: typically represent things that are to be stored in a database and persist throughout the duration of the application

Boundary classes are designed with the responsibility of managing the way entity objects are represented to users. For example, a boundary class called **CameraWindow** would have the responsibility of displaying surveillance camera output for the *SafeHome* system

In general, controller classes are not considered until the design activity has begun.



CRC: Responsibility

- How to assign responsibilities to classes
  - System intelligence should be distributed across classes to best address the needs of the problem
  - Each responsibility should be stated as generally as possible
  - Information and the behavior related to it should reside within the same class
  - Information about one thing should be localized with a single class, not distributed across multiple classes.
  - Responsibilities should be shared among related classes, when appropriate.

Basic guidelines for **identifying** responsibilities (attributes and operations) have been presented in previous slides.

Every application encompasses a certain degree of intelligence; that is, what the system knows and what it can do.

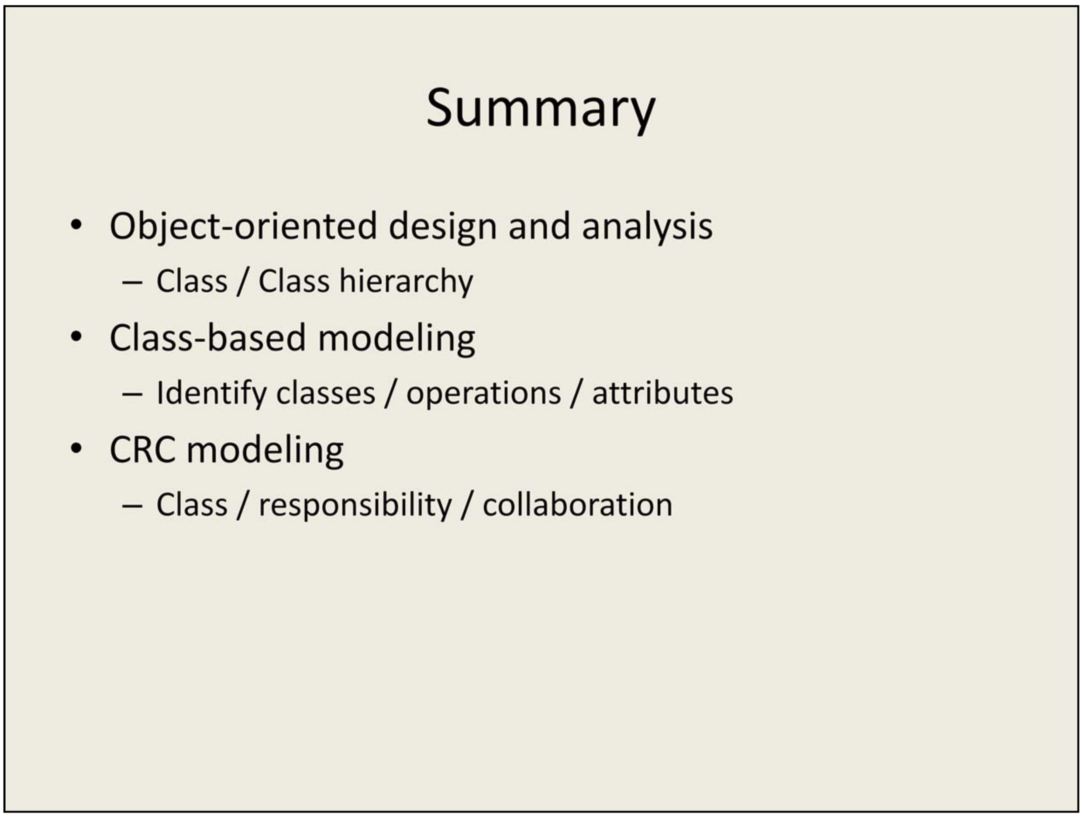
If system intelligence is more evenly distributed across the classes in an application, each object knows about and does only a few things (that are generally well **focused**), the cohesiveness of the system is improved. This enhances the maintainability of the software and reduces the impact of side effects due to change.

This guideline implies that general responsibilities (both attributes and operations) should reside high in the class hierarchy (because they are generic, they will apply to all subclasses).

As an example, consider a video game that must display the following classes: **Player, PlayerBody, PlayerArms, PlayerLegs, PlayerHead.** Each of these classes has its own attributes (e.g., position, orientation, color, speed) and all must be updated and displayed as the user manipulates a joystick. The responsibilities *update()* and *display()* must therefore be shared by each of the objects.

CRC: Collaboration

- Classes fulfill their responsibilities in one of two ways:
  - A class can use its own operations to manipulate its own attributes, thereby fulfilling a particular responsibility, or
  - a class can collaborate with other classes.
- Collaborations identify relationships between classes
- Collaborations are identified by determining whether a class can fulfill each responsibility itself
- three different generic relationships between classes:
  - the *is-part-of* relationship
  - the *has-knowledge-of* relationship
  - the *depends-upon* relationship



Summary

- Object-oriented design and analysis
  - Class / Class hierarchy
- Class-based modeling
  - Identify classes / operations / attributes
- CRC modeling
  - Class / responsibility / collaboration